Indice

Voci

Pascal

Pascal

Il Pascal è un linguaggio di programmazione inizialmente basato sul concetto di programmazione strutturata, caratterizzato da una estrema rigidità, poi esteso anche alla programmazione orientata agli oggetti.

Le sue peculiarità sono date dall'intenzione del suo creatore, Niklaus Wirth, di creare un linguaggio che, a differenza del BASIC, forzasse da parte del programmatore uno studio accurato dell'algoritmo e che fosse dotato di funzionalità e strutture dati avanzate.
Questi elementi lo rendono uno dei linguaggio più usati a livello didattico, soprattutto nelle scuole.

Finalità

Il libro non richiede particolari conoscenze di programmazione per essere letto. È stato pensato per seguire il lettore dalle "basi" della programmazione fino ad una conoscenza non avanzata del linguaggio Pascal.

Altri progetti

- **Wikipedia** contiene una voce sul **linguaggio Pascal**

Introduzione

Il **Pascal** è un linguaggio di programmazione nato nel 1973 ad opera di *Niklaus Wirth*.
Egli, oltre ad essere un informatico, era anche un insegnante di programmazione: creò il Pascal proprio come linguaggio didattico e lo progettò appositamente molto rigido; questo perché richiedesse prima dell'implementazione un'attenta analisi da parte dell'alunno dell'algoritmo da sviluppare.
Il Pascal è inoltre un linguaggio piuttosto facile da usare, anche per la sua somiglianza con la lingua inglese che rende alcuni comandi piuttosto intuitivi.

Ancora oggi, il Pascal è il linguaggio più usato a livello didattico; per questo motivi è utilizzato diffusamente come linguaggio introduttivo alla programmazione strutturata. Questo non significa però che abbia poche potenzialità. Wirth pensava al Pascal come un linguaggio facile ma potente; addirittura la sua evoluzione orientata agli oggetti, il Delphi, è estremamente diffusa in vari ambiti.

Hello world!

Storicamente, ogni volta che si deve presentare un linguaggio di programmazione, si utilizza il classico *Hello World*, un piccolo programma che stampa sullo schermo, appunto, *Hello World!*. In Pascal la sua realizzazione è piuttosto semplice:

```
program HelloWorld;
begin
     writeln('Hello world!');
end.
```

- la prima riga indica il nome del programma. La parola *program* è una parola riservata del linguaggio. Non può quindi essere usata come nome di variabile o di funzione (che vedremo più avanti nel corso del libro). Alla fine di ogni istruzione bisogna inserire il carattere *;* per indicare al compilatore che il comando è concluso.
- la seconda riga indica l'inizio del programma vero e proprio. Ogni blocco di codice deve essere inserito fra le parole riservate `begin` e `end`.
- la terza riga contiene la nostra prima vera istruzione. I comandi di input-output verranno trattati in seguito. Per ora basti sapere che l'istruzione `writeln` permette di mostrare sullo schermo i caratteri inseriti fra gli apici.
- l'ultima riga rappresenta la conclusione del programma. La parola riservata *end* posta alla fine del programma deve essere sempre seguita dal punto.

Per alcuni compilatori (soprattutto in ambiente Windows) sarà necessario aggiungere un'ultima riga di codice, **`readln;`**, prima di `end`. L'istruzione infatti blocca l'esecuzione del programma fino a che questi non preme <Enter>, permettendo così all'utente di vedere l'output del programma prima che si chiuda la finestra in cui è stato fatto partire. L'istruzione `readln;` verrà comunque trattata approfonditamente più avanti nel corso di questo libro.

Per avere delle istruzioni su come effettuare la compilazione, consulta questo modulo.

Primi concetti sulla sintassi

Dal nostro primo esempio possiamo ricavare alcune semplici indicazioni sulla sintassi di un programma implementato in Pascal:

- esistono delle **parole riservate**, che fanno parte della grammatica del linguaggio Pascal, che possono essere usate solo con lo scopo prefissato.
- ogni programma è composto da **istruzioni**, semplici o composte.
- le istruzioni vengono eseguite dalla prima all'ultima in **sequenza**.
- le istruzioni composte vengono detti **blocchi**. Vengono individuati da una coppia di parole chiave: *begin* indica l'inizio del blocco, *end* la fine e contengono a loro volta istruzioni.
- il blocco può essere considerato come un'unica istruzione.
- ogni istruzione è separata da un carattere speciale: *;* il punto e virgola.
- esistono degli identificatori, ovvero nomi arbitrari per variabili, costanti, procedure e funzioni (che vedremo più avanti).

Algoritmi

Prima di continuare ad analizzare le strutture del linguaggio Pascal, ci soffermeremo su un concetto molto importante in programmazione, e che assume ancora più significato con questo linguaggio: quello di **algoritmo**.

Il linguaggio Pascal è stato infatti creato molto rigido per costringere il programmatore ad un'accurata analisi del problema (e quindi dell'algoritmo).

Cos'è un algoritmo?

Un **algoritmo**, nel suo significato più ampio, è sequenza logica di istruzioni elementari (univocamente interpretabili) che, eseguite in un ordine stabilito, permettono la soluzione di un problema in un numero finito di passi

Dal punto di vista dell'informatica, possiamo quindi considerare ogni nostro programma un algoritmo, dove ciascuna istruzione è uno dei passi dell'algoritmo. Analizzare gli algoritmi ci aiuterà quindi a capire meglio come funziona la programmazione e, più in particolare, il concetto di programmazione strutturata su cui si basa il Pascal.

Vedremo inoltre come tutte le strutture possono essere rappresentate sotto forma di diagrammi a blocchi, il cui uso è molto comodo per analizzare gli algoritmi e lo sviluppo di programmi complessi.

Il teorema di Böhm-Jacopini

Il teorema di Böhm-Jacopini, enunciato nel 1966 dai due informatici dai quale prende il nome, afferma che:

« qualunque algoritmo può essere implementato utilizzando tre sole strutture, la sequenza, la selezione ed il ciclo, da applicare ricorsivamente alla composizione di istruzioni elementari »

(teorema di Böhm-Jacopini)

Questo teorema in sostanza dice che ciascun algoritmo (e quindi, come abbiamo detto, ciascun programma) può essere realizzato utilizzando soltanto le tre strutture indicate, che ci appresteremo ad analizzare: la sequenza, la selezione (binaria o multipla) ed il ciclo (o ripetizione). Per struttura di controllo si intende una struttura sintattica che determina come e se devono essere eseguite certe istruzioni.

Le strutture di controllo

Ciascuna istruzione di controllo in un linguaggio di programmazione strutturata ha alcune caratteristiche:

- ha un solo punto di ingresso e punto di fine
- la componibilità: ogni struttura di controllo può avere tra le strutture che controlla altre strutture di controllo, ricorsivamente, senza limiti.

La sequenza

La struttura della **sequenza** è la più semplice, ed è stata già vista in precedenza: prevede che le istruzioni siano ripetute una dopo l'altra.
Nel caso di un semplice algoritmo "prepara il caffè", i passaggi in ordine saranno ad esempio prendi il caffè, riempi il filtro della caffettiera, accendi il gas, ecc...

La selezione

La struttura di **selezione** permette di eseguire blocchi di istruzioni differenti in base ad una condizione valutata inizialmente.
Nel nostro algoritmo "prepara il caffè" potremmo quindi ad esempio prevedere azioni differente se non c'è caffè in casa (si va a comprare il caffè); comunque, anche questa selezione poi ha un unico punto di uscita, cioè riempi il filtro della caffettiera, sia che ci sia il caffè in casa o no.
Questo è un caso di *selezione binaria*, che prevede cioè solo due alternative: il caffè è in casa, oppure non lo è.

Talvolta è possibile prevedere anche più di un caso, ad esempio che tipo di caffè si vuole preparare (cappuccino, macchiato, semplice, ecc...): in base all'alternativa scelta, bisogna effettuare operazioni differenti. In realtà questo costrutto (chiamato *selezione multipla*) può essere sostituito da più selezioni binarie.

Il ciclo

Alcune operazioni possono poi richiedere la **ripetizione** di una o più istruzioni: questo è possibile grazie alla struttura chiamata "ciclo" o "ripetizione" o anche "iterazione" (dal verbo latino *itero*, ripetere).
Nel nostro algoritmo "prepara il caffè" ad esempio, si può inserire un ciclo per il numero di cucchiaini di zucchero necessari: se sono necessari 3 cucchiaini di zucchero, allora si dovrà ripetere l'azione "versa zucchero" per tre volte.

Componibilità

L'algoritmo appena descritto può essere reso più fine, prevedendo ad esempio al posto dell'operazione "vai a comprare il caffè" un *sotto-algoritmo* (in programmazione, un *sotto-programma*) che preveda la scelta del negozio da comprare in base ad esempio alla convenienza del caffè, ecc...

Input e output

Altri due importanti concetti su cui soffermarsi prima di affrontare lo studio del Pascal sono quelli di **input** e **output**.

Per ***input*** si intendono tutte le informazioni *in ingresso* nell'algoritmo, che vengono cioè dall'esterno, e non sono state definite da chi ha formulato l'algoritmo (o il programma; in effetti, si parla di input e output più in informatica che in altri campi). In informatica ad esempio è un input anche il muovere il mouse, cliccare su un pulsante o digitare dei caratteri da tastiera.
Nel nostro algoritmo "prepara il caffè", ad esempio, sono informazioni in input il numero di tazzine da preparare, il tipo di caffè che hanno scelto, ecc...

L***'output*** costituisce invece tutte le informazioni *in uscita* dall'algoritmo, cioè quello che produce effettivamente l'algoritmo sull'esterno. In informatica, tipicamente un output può essere un risultato stampato a schermo, oppure una finestra che ci avverte di un errore o di un'operazione avvenuta. A parità di input, l'output deve essere il medesimo.
Nel nostro algoritmo "prepara il caffè" può essere considerati ouput *in primis* il caffè che abbiamo preparato!

Variabili

Un programma come quello appena fatto può essere divertente e interessante, ma offre ben poche possibilità di sviluppo. Quasi tutti i tipi di programmi richiedono infatti l'uso di calcoli e di "valori da ricordare". Per questo motivo è necessario introdurre il concetto di **variabile**.

Una variabile può essere paragonata ad una casella di una certa dimensione nella quale si possono inserire e/o leggere dati mentre il programma è in esecuzione. In pratica le variabili sono delle etichette che noi diamo ad un'area di memoria, che verrà usata per depositare dei dati. La quantita di memoria riservata dipende dal tipo di variabile che andiamo a dichiarare e dal particolare compilatore che usiamo.

Il compilatore Pascal ha la necessita di conoscere l'uso e lo scopo di tutte le etichette che incontra durante la fase di compilazione, è indispensabile quindi dichiarare esplicitamente, in particolari punti del programma, le variabili e altro che vedremo. L'area di dichiarazione delle variabili inizia con la parola chiave *var*.
Il concetto risulta più chiaro con un esempio:

```
program VariabiliVarie;
var
    n:integer;
    r:real;
begin
     n:=3;
     r:=sqrt(n);
     n:=5;
     writeln(n);
     writeln(r);
end.
```

Il programma in sè è estremamente sciocco, ma ci permette di osservare come le variabili vengano utilizzate in Pascal. Analizzamo le singole righe:

- la prima riga è l'intestazione del programma (alla parola riservata *program* segue il nome scelto per il programma)
- nella seconda e nella terza riga vengono dichiarate le variabili. La dichiarazione ha sempre la forma

```
var
    nomevariabile : tipo;
    nomevariabile2 : tipo2;
```

Il Pascal non permette di dichiarare durante il programma variabili aggiuntive, perciò è necessaria una buona progettazione teorica del programma per non trovarsi a dover correggere molti errori in corso di compilazione (ovviamente ciò era tra le intenzioni di Wirth quando progettò il linguaggio).

Assegnazione

Una delle operazioni fondamentali che si possono eseguire con le variabile è l'istruzione di assegnazione.
Per assegnare si intende attribuire ad una determinata variabile un valore specificato dal programmatore oppure il risultato di un'espressione (per vedere come operare con le variabili, leggi il prossimo modulo)

La **sintassi** dell'assegnazione è piuttosto semplice:

```
nome_della_variabile := valore_assegnato_alla_variabile;
```

dove valore_assegnato_alla_variabile è un valore o un'espressione il cui risultato è del tipo di dato dichiarato all'inizio del programma. Se infatti si tenta di assegnare un valore del tipo *x* ad una variabile del tipo *y* il compilatore restituisce un errore.

Altri concetti di sintassi

Abbiamo visto due esempi. Possiamo notare che entrambi riflettono anche uno stile di scrittura del codice, che diciamo **stile di formattazione**. Notiamo che il codice presenta varie rientranze. Queste hanno lo scopo di rendere più semplice la vita del programmatore. Consentono di individuare a colpo d'occhio un blocco, semplicemente osservando l'andamento delle rientranze.
Per il compilatore tutto ciò è ininfluente, in quanto lui individua i blocchi tramite le parole chiavi e trova le istruzioni separate da un carattere apposito (il punto e virgola).

Se provate a compilare le seguenti righe otterrete lo stesso programma visto nell'esempio precedente. Per il compilatore non ci sono differenze, mentre per il programmatore che dovrà estendere o correggere questo codice le cose si complicano.

```
program VariabiliVarie;
var n:integer; r:real;
begin
      n:=3;
      r:=sqrt(n);
      n:=5;
      writeln(n);
      writeln(r);
end.
```

Per lo stesso motivo anche un programma come questo è valido:

```
program VariabiliVarie;  var n:integer; r:real; begin n:=3; r:=sqrt(n);
 n:=5; writeln(n); writeln(r); end.
```

ma è ovviamente molto meno leggibile.

Ma quali sono i tipi di variabile utilizzabili? Un elenco esaustivo è presente nel modulo Tipi di dati

Input e output

Finalmente ci occupiamo del grande problema dell'**input-output**.
Gran parte delle funzioni di input-output sono contenute nella libreria standard di Pascal. Queste funzioni sono tipicamente:

- *read()* e *readln()* che permettono di leggere l'input dell'utente da tastiera inserendo fra le parentesi il nome della variabile in cui vogliamo salvare il dato. La differenza consiste nel cursore, che nel caso di un'istruzione *read* continua sulla stessa riga mentre nel caso di *readln* va a capo.
- *write()* e *writeln()* che permettono, come già visto in precedenza, di stampare su schermo il contenuto delle parentesi. Anche in questo caso la differenza è relativa all'andare a capo.

Un semplice esempio per iniziare

Un semplice esempio basato sull'uso delle variabili e dell'input-output potrebbe essere un modo per personalizzare un programma, salvando in una stringa il nome dell'utente per inserirlo nelle domande.

```
program username;
     uses crt;
     var name: String[50];

begin
    clrscr;
    writeln('Inserisci il nome');
    readln(name);
    clrscr;
    writeln('Benvenuto, ', name);
    readkey;
 end.
```

Analizziamo come al solito il listato riga per riga:

- dichiarazione del nome
- dichiarazione della variabile *name* di tipo Stringa di dimensione 50 (i tipi di dati saranno affrontati in seguito)
- dichiarazione delle librerie necessarie. Di questo argomento riparleremo a breve, ma sappiate che almeno due dei comandi utilizzati di seguito sono dipendenti da questa libreria
- inizio programma
- pulizia dello schermo. Questa è la prima (e forse la più usata) delle due funzioni che dipendono da *crt*.
- stampa sullo schermo della stringa 'Inserisci il nome'. Dopo questa istruzione il programma va a capo
- lettura della stringa corrispondente al nome ed inserimento di questa stringa letta da tastiera nella variabile *name*
- nuova pulizia dello schermo
- stampa del messaggio di benvenuto
- altra funzione dipendente dalla libreria *crt* che permette di leggere un solo carattere qualunque da tastiera.
  Vedremo che questa funzione è molto utilizzata per risolvere un problema riguardo all'esecuzione dei programmi compilati finora.

read e readln

L'istruzione assegna il valore immesso dall'utente nella variabile specificata tra parentesi. Il contenuto delle parentesi che seguono l'istruzione *read* o *readln* può essere una o più variabili. Nel secondo caso, in fase di esecuzione, i diversi valori di input vanno inseriti separandoli con uno spazio.

write e writeln

Il contenuto della parentesi che seguono l'istruzione *writeln* può essere sia un'espressione (stringa o numerica), sia una variabile sia una combinazione.
La parte testuale è compresa tra due apici; per separare il testo dalle variabili si usa una virgola, come nell'esempio precedente.

Alcuni compilatori, soprattutto i più vecchi, possono inoltre dare problemi nella stampa dei caratteri accentati: in questi casi l'accento può essere sostituito con un apostrofo che, per il problema relativo agli apici nelle stringhe, si deve indicare con due apici. Un'istruzione come

```
writeln('Il valore è ');
```

è preferibile sostituirla quindi con

```
 writeln('Il valore e''');
```

(che stampa *Il valore e*').

Formattare i numeri

Quando la variabile inserita nel *writeln* è di tipo numerico è possibile specificare con quante cifre dev'essere rappresentato il numero e quante di queste devono essere decimali (nel caso il numero sia reale e non intero); per fare ciò si utilizza la sintassi

```
nome_della_variabile : numero_di_cifre [: numero_di_cifre_decimali];
```

Ad esempio:

```
writeln(n:8:6);
writeln(10+2:4);
```

Ovvero, nel primo caso il contenuto della variabile *n* verrà rappresentato con 8 cifre di cui 6 decimali; nel secondo caso verranno stampati 2 spazi vuoti e poi il numero 12.
Una istruzione come questa

```
writeln(n);
```

stamperebbe invece il valore di *n* in notazione esponenziale (ad esempio 3.4E4 anziché 34.000).

Riguardo input e output, si può anche fare un esempio con i numeri di con un programma che calcoli il quadrato di un numero reale:

```
program quadrato_di_un_numero;
uses crt;
var n,r:real;
begin
    clrscr;
    writeln('Inserisci un numero reale');
    readln(n);
```

```
    clrscr;
    r:=sqr(n);
    writeln('Il quadrato del numero inserito è',r:10:6);
    readkey;
end.
```

- il programma stampa la stringa 'Inserisci un numero reale';
- con *readln(n)* assegna alla variabile n un valore immesso da tastiera;
- alla variabile *r* viene assegnato il valore del quadrato di *n*;
- viene scritto il risultato.

Si noti che per l'assegnazione di un certo valore a una variabile si usa la sintassi:

```
nome_della_variabile := valore_assegnato_alla_variabile;
```

Questo valore può essere, come nell'esempio precedente, il risultato di un'espressione.

Esercizi

- Scrivere un programma che applichi l'*algoritmo dello scambio*. Il programma deve chiedere all'utente i valori da inserire in due variabili, scambiare il contenuto delle variabili e poi stamparle (nello stesso ordine in cui le ha chieste). L'utente vedrà così scambiato l'ordine dei valori inseriti.
- Scrivere un programma che legga un numero e lo stampi formattandolo, chiedendo all'utente quante cifre utilizzare per i decimali.

Tipi di dati e operatori

In questo modulo analizzeremo i tipi di dati nativi di Pascal (che sono predefiniti e non richiedono l'uso di particolari librerie.

Ricordiamo che la dichiarazione di una variabile segue questa sintassi:

```
var
   nome : tipo;
```

Integer

Il tipo di dato più utilizzato è in genere il tipo *integer*. Corrisponde ai numeri interi, anche se non proprio a tutti; infatti, per via di problemi di memoria, non può memorizzare tutti i numeri: in Turbo Pascal comprende gli interi da -32768 a 32767 (in questo modo una variabile integer occupa una memoria di 16 bit - 2 byte), ma in altre versioni questo numero può essere maggiore.
Essendo il computer una macchina limitata, infatti, non è possibile fare somme fino all'infinito. Con alcuni semplici programmi si può osservare che esiste un limite superiore per gli interi , oltre al quale si ricade nel limite inferiore, come se fosse una circonferenza. Il fenomeno dello sforamento del limite massimo è detto overflow.

Ogni tipo di dato ha delle operazioni permesse. In questo caso le operazioni che danno come *risultato* un integer sono:

- + permette di sommare due valori di tipo integer.
- - permette di calcolare la differenza fra due valori integer.
- * permette di moltiplicare due valori, ottenendo sempre un integer.
- *div* permette di calcolare la parte intera del rapporto fra due interi.

- *mod* permette di calcolare il resto della divisione fra due interi.

Le funzioni matematiche che danno come risultato un intero sono, invece

- *abs(n)* che calcola il valore assoluto del numero intero *n*
- *round(n)* che arrotonda qualunque numero all'intero più vicino ad *n*
- *trunc(n)* che tronca il numero *n* alla parte intera (ad esempio *trunc(3.7)* restituisce 3, ma *trunc(-3.7)* restituisce ancora 3)
- *sqr(n)* ne calcola infine il quadrato.

Proviamo a fare un programma che utilizzi alcune operazioni. Ad esempio, il seguente programma valuta se un numero è pari o dispari:

```
program Pari;
var n:integer;
begin
     readln(n);
     if (n mod 2 = 0) then write ('pari')
                      else write ('dispari');
     readln;
end.
```

Il programma presenta alcuni costrutti che non abbiamo ancora visto, ma la cui interpretazione è semplice. Analizziamo come al solito riga per riga.

- Come al solito la prima riga contiene la dichiarazione del titolo.
- Dichiarazione di una sola variabile *n* di tipo intero.
- Inizia il programma.
- Assegniamo a *n* il valore 5.
- Inizia un costrutto condizionale. Letto "alla lettera" significa "se il resto della divisione fra n e 2 è uguale a zero, allora scrivi *pari* altrimenti scrivi *dispari*". Da notare, lo rivedremo e faremo alcune considerazioni, che prima di *else* non va **mai** messo il ";".

Char

Frequentemente si utilizzano invece degli *integer* i *char*, per rappresentare numeri interi piccoli. Infatti i *char* rappresentano il range di numeri interi che va da 0 a 255 - corrispondente ai codici ASCII - e permettono la visualizzazione nei due formati: intero, ascii. Le operazioni possibili sono le stesse degli integer, ma ci sono due funzioni in più che permettono la trasformazione da intero a carattere e viceversa. Queste funzioni sono

- *chr(x)* che permette di trasformare l'intero *x* nel corrispondente carattere
- *ord(x)* che permette di trasformare il carattere *x* nel corrispondente intero ASCII

Un semplice esempio di uso potrebbe essere la stampa della tabella dei codici ASCII.

```
program ASCII;
var n:integer;
begin
     for n:=0 to 255 do
     writeln(n, ' ==> ', chr(n));
end.
```

Come al solito analizziamo riga per riga.

- Dichiarazione del programma,

- Definizione della variabile *n* come char
- Inizio del programma
- Il costrutto *for...to...do* non l'abbiamo ancora visto, ma facendo come prima la traduzione letterale, possiamo intuirne il significato: *Per n che va da 0 a 255 fai...* È la traduzione in Pascal della struttura della *ciclo* o *iterazione* già vista nel modulo sugli algoritmi.
- scriviamo su schermo il numero *n* seguito dai caratteri ==> e poi dal corrispondente carattere ASCII.

Per assegnare ad una variabile *char* un valore è necessario usare la notazione

```
nome_variabile := 'carattere';
```

I due caratteri ' (carattere apice) identificano un valore alfanumerico e non numerico.

Real

Le variabili di tipo real corrispondono ai numeri reali, ovvero i numeri con la virgola. Anche qui, come per gli integer, dobbiamo dare delle limitazioni. Essendo, come già sottolineato un computer una macchina limitata e discreta, inevitabilmente non può superare una certa precisione nel calcolo con i numeri reali. Per questo motivo spesso nelle strutture condizionate si devono cercare di utilizzare stratagemmi per evitare problemi.

Le operazioni di base possibili sembrano meno di quelle possibili con char e integer, ma attraverso l'utilizzo della libreria matematica aumentano in modo incredibile. Comunque le operazioni basilari sono:

- + permette di sommare due valori di tipo real.
- - permette di calcolare la differenza fra due valori real.
- * permette di moltiplicare due valori, ottenendo sempre un real.
- / permette di calcolare il rapporto fra due real.

Interne all'insieme dei reali troviamo ad esempio funzioni come:

- *sin(a)* e *cos(a)*: restituiscono il seno e il coseno dell'angolo a (espresso in radianti)
- *random*: fornisce un numero casuale compreso tra 0 e 1 esclusi. Prima di usare questa funzione è bene usare *randomize*, che reinizializza il generatore di numeri casuali.

Boolean

L'algebra booleana è fondamentale nell'informatica. Questa permette infatti di fare calcoli sulla veridicità o falsità di una affermazione. Le variabili booleane infatti possono assumere solo i valori logici vero (*true*) e falso (*false*). Le operazioni possibili sono diverse da quelle possibili per gli altri tipi di variabile:

- *and* che corrisponde al simbolo matematico $\wedge$ e al concetto di *e contemporaneamente*
- *or* che rappresenta $\vee$ e *oppure*
- *not* operatore che corrisponde alla negazione
- *xor* che corrisponde matematicamente all'*aut* (simbolo matematico $\dot{\vee}$), cioè il concetto di *o uno, o l'altro, ma non tutti e due*.

Le variabili di tipo booleano si dichiarano con la parola riservata *boolean*.

```
var
 variabile_booleana:boolean;
```

Si inizializzano così:

```
variabile_booleana:=true;
{oppure}
variabile_booleana2:=false;
```

```
variabile_2 := variabile_booleana or variabile_booleana2;
variabile_3 := true or variabile_booleana2;
variabile_4 := true xor false;
{ma anche}
num_pari := (n mod 2 = 0);
```

Potrebbe non essere immediata l'ultima espressione; il suo significato è: assegna alla variabile `num_pari` il valore dell'espressione `n mod 2 = 0`; questa espressione contiene un segno di uguaglianza (che è un operatore di confronto) e quindi restituisce *vero* o *falso*. L'utilizzo di variabili booleane è in genere limitato all'analisi di particolari aspetti dell'input da utente, o all'utilizzo come 'flag' nella programmazione più avanzata.

String

Le variabili string sono variabili che possono contenere una stringa alfanumerica di caratteri.

La dichiarazione di una variabile string ha sintassi:

```
nome_della_variabile : string[n];
```

Ovvero afferma che la variabile *nome_della_variabile* può contenere una riga di massimo *n* caratteri.

Come per le variabili *char*, anche le variabili *string* necessitano una sintassi particolare per l'assegnazione: l'istruzione

```
nome_variabile := 'Ma la volpe col suo balzo ha raggiunto il quieto
fido 1234';
```

assegna a nome variabile la stringa alfanumerica (in questo caso *Ma la volpe col suo balzo ha raggiunto il quieto fido 1234*.

Nel caso si voglia inserire un carattere apice nella propria stringa (per inserire ad esempio un apostrofo) è necessario usare due apici di seguito. Un'istruzione come questa:

```
writeln('L'unica soluzione...');
```

genera infatti un errore, poiché l'apostrofo tra *La* e *unica* è interpretato dal compilatore come un apice che delimita la fine della stringa. Un'istruzione invece come questa

```
writeln('L''unica soluzione...');
```

è interpretata correttamente.

L'unico operatore possibile con il tipo string è l'operatore di concatenazione +, che è possibile capire con un esempio:

```
var1 := 'Ma la volpe col suo balzo ';
var2 := 'ha raggiunto il quieto fido';
var3 := var1 + var2 + ' 1234';
```

La variabile var3 alla fine di questo breve listato conterrà così il valore *Ma la volpe col suo balzo ha raggiunto il quieto fido 1234*, in quanto le stringhe sono state concatenate in una stringa unica.

Si noti lo spazio inserito prima dell'apice: senza lo spazio, le parole risulterebbero attaccate.

Una funzionalità interessante relativa alle stringhe è l'**indicizzazione**, per la quale è possibile fare riferimento ad un carattere all'interno della stringa, usando la notazione

```
variabile[n] {l'n-esimo carattere della stringa in variabile}
```

Riguardo alle stringhe, ricordiamo la funzione `length` che restituisce la lunghezza di caratteri della stringa passata come paramero.

Riassumendo

Operazioni, operatori e funzioni

Operatore	Operandi	Risultato
+	real o integer o char	real o integer o char
-	real o integer o char	real o integer o char
*	real o integer o char	real o integer o char
/	real o integer o char	real
mod	integer o char	integer
div	integer o char	integer
Funzione	**Argomento**	**Risultato**
sqr(*x*)	real o integer o char	integer
sqrt(*x*)	real o integer o char	real
abs(*x*)	real o integer o char	real o integer o char
trunc(*x*)	real o integer o char	integer
sin(*x*) e cos(*x*)	real o integer o char	real
random	-	real

Il significato di questi operatori e di queste funzioni è stato mostrato nei paragrafi precedenti. ad eccezione della funzione *sqrt(*n*)*, che restituisce la radice quadrata di un *n*.

Operatori di confronto

Pascal mette a disposizione inoltre alcuni operatori di confronto che restituiscono sempre un valore booleano e che possono essere usati su due variabili qualsiasi dello stesso tipo di dato:

Gli operatori di confronto in Pascal

<	minore
<=	minore o uguale
=	uguale
<>	diverso - si può usare anche not(val1 = val2)
>=	maggiore o uguale
>	maggiore

È intuibile il loro funzionamento nell'ambito di variabili integer o real; questi operatori tuttavia possono essere anche usati sui tipi di dato char o string; più in particolare:

```
char1 < char2
```

è equivalente a scrivere

```
ord(char1) < ord(char2)
```

La comparazione tra valori string valuta invece le stringhe in ordine alfabetico (quindi saranno vere le espressioni come, ad esempio, 'abaco' < 'amico' o 'zaino' = 'zaino').

Per quanto riguarda i valori boolean, vale

```
false < true
```

e, quindi

```
true > false
```

Estensioni dei dati nativi

Alcuni compilatori moderni forniscono delle estensioni dei dati nativi, per ovviare ad esempio alla limitatezza degli *integer*: un esempio è il tipo di dato *longint*, disponibile in alcuni compilatori, che ha le stesse caratteristiche degli *integer* ma ammette numeri in un intervallo molto maggiore.

Esercizi

- Scrivere un programma che prenda una misura di tempo in secondi e la scriva in giorni, ore, minuti e secondi.
- Scrivere un programma che faccia il contrario, converta cioè una misura espressa in giorni, ore, minuti e secondi in secondi.
- Scrivere un programma "dado" che simuli il lancio di un dado usando la funziona *random*.
- Scrivere un programma che legga una valùta dall'utente e converta la misura in euro considerando la valute espressa in lire, e viceversa.

Gestione avanzata dei dati

In questo capitolo tratteremo tipi di dati più complessi e che utilizzeremo meno frequentemente

Tipi di dati definiti dall'utente

Abbiamo già visto prima che per dichiarare una variabile si deve specificare un tipo (ad esempio, integer o real) per indicare quali valori può assumere.
In alcuni casi può essere utile ricorrere a tipi di dati non predefiniti, ma creati dall'utente: si può ricorrere quindi ad una *dichiarazione di tipo* con la parola riservata ***type***.
La sintassi è

```
type
 nome: valori_possibili;
```

che definisce il tipo di dato *nome*.
Per indicare quindi che una variabile è del tipo *nome* si usa la sintassi:

```
var
 variabile:nome;
```

Tipi enumerativi

Il più semplice tipo di dichiarazione *type* è quella dei **tipi enumerativi**, che consentono cioè un numero ristretto di valori, definiti singolarmente dall'utente:

```
type nome:(elemento_1, elemento_2,...,elemento_n);
```

dove elemento_1,elemento_2,...,elemento_n sono i possibile valori che possono assumere le variabili del tipo di dato *nome* elencati in ordine.
Per fare un esempio, possiamo definire un tipo:

```
type mesi: (gen, feb, mar, apr, mag, giu, lug, ago, sett, ott, nov,
dic);
```

e dichiarare due variabili

```
var mese_di_nascita_luigi, mese_di_nascita_gianni:mesi;
```

Se nel corso del programma poniamo

```
mese_di_nascita_luigi:= feb;
mese_di_nascita_gianni:= ago;
```

avremo che: l'istruzione `writeln(mese_di_nascita_luigi);` stampa sullo schermo *feb*; l'espressione `mese_di_nascita_luigi < mese_di_nascita_gianni` restituisce TRUE, in quanto feb precede ago nella lista dei mesi indicata nella dichiarazione. Allo stesso modo `giu > set` restituisce invece FALSE.

Subrange

I ***subrange*** sono tipi di dati che sono definiti come sottoinsiemi di altri tipi. Ad esempio:

```
type
    cifra: 0..9
    mese_primaverile: mar..giu;
    lettera: 'a'..'z';
```

Abbiamo definito tre dati subrange: il primo tipo permette tutti i numeri dallo 0 al 9, il secondo tutti i mesi primaverili (da marzo a giugno) e il terzo tutti i caratteri da "a" a "z". Come si può intuire:

- una definizione di un subrange è costituita da un **limite** inferiore e uno superiore
- un subrange non è valido se il limite inferiore è maggiore o uguale a quello superiore (ad esempio non è valido il subrange `9..5`)
- i due limiti devono essere dello stesso tipo di dato (interi, *char*, o anche definiti dall'utente, come nel caso di `mese_primaverile` che è un subrange del tipo *mesi* definito in precedenza)

Una volta definito, lo possiamo utilizzare:

```
var mese: mese_primaverile;
```

In questo caso ad esempio si possono effettuare dichiarazioni come `mese := mar;` ma non `mese := ott;`.

Insiemi

Vediamo ora un altro tipo di dato non molto utilizzato, ma che può risultare comodo in alcune occasioni: gli **insiemi**, che in Pascal sono molto simili al concetto di insieme in matematica; dato un insieme A di oggetti di un insieme di dati B, ogni oggetto di B appartiene o non appartiene ad A.

Vediamo come dichiarare gli insiemi con un esempio:

```
var giorni_liberi: SET OF integer;
```

Possiamo altrimenti scrivere:

```
type giorni_del_mese: SET OF integer;
var giorni_liberi : giorni_del_mese;
```

Ma il risultato è lo stesso. Per inizializzare un insieme usiamo la sintassi:

```
giorni_liberi := [1,3..7, 10..25, 29];
giorni_liberi := []; {insieme vuoto}
```

Si noti l'uso di un *range* nell'assegnazione. La sintassi generale è quindi per un insieme `[el1, el2, el3..el7, ...]`.

Operazioni con gli insiemi

Le operazioni fattibili con gli insiemi sono le stesse in matematica: ad esempio, "+" corrisponde all'unione, "-" alla differenza e "*" all'intersezione.

```
[1, 3, 4] + [3, 9..11] = [1, 3, 4, 9..11]
[5, 7, 19, 22] - [22] = [5, 7, 19]
[5, 7, 19, 22] - [23] = [5, 7, 19, 22]
[11..22] * [15..56] = [15..22]
[11..22] + [15..56] = [11..56]
[5, 7, 19, 22] * [22] = [22]
[5, 7, 19, 22] * [23] = []
```

Per verificare se un elemento è in un insieme, usiamo l'operatore IN:

```
if 1 in giorni_liberi then
   ...
```

È possibile anche confrontare gli insiemi:

- **=**: restituisce *true* se i due insiemi sono uguali;
- **<>**: restituisce *true* se i due insiemi sono diversi;
- **<=**: restituisce *true* se il primo insieme è sottoinsieme del secondo (il secondo contiene il primo);
- **>=**: contrario di <=

Commenti

In programmazione, un **commento** è una parte del listato che viene ignorata dal compilatore, ma che è funzionale a chi ha steso il codice al fine di lasciare qualche appunto, specie se il codice in questione deve essere letto da altri programmatori.

In Pascal i commenti si indicano delimitandoli da parentesi graffe (ottenibili tramite Shift+Alt Gr+tasto delle parentesi quadre), o da parentesi tonde seguite e precedute da un asterisco; ad esempio:

```
{Questo è un commento}
```

oppure

```
(*Questo è un commento*)
```

L'utilità dei commenti è evidente nei casi di scambio dei sorgenti di un programma all'interno di una comunità di sviluppatori: è spesso difficile riuscire infatti a capire il funzionamento di un algoritmo studiato e codificato da qualcun altro senza l'ausilio di commenti scritti in maniera efficace.

I commenti possono rivelarsi utili inoltre nel caso un autore di un programma riprenda il suo lavoro dopo un periodo di pausa: nel caso il codice non fosse ben commentato, infatti, l'autore si ritroverebbe nella stessa situazione di un qualsiasi programmatore che non ha mai letto il codice e gli sarebbe quindi difficile riprendere il lavoro in maniera adeguata.

Librerie e funzioni predefinite

Una **libreria** non è altro che un file contenente varie procedure, funzioni e/o costanti che possono essere utilizzate una volta che il file è stato incluso all'interno del programma. Questo da un punto di vista teorico è molto utile perché permette di avere programmi molto brevi, ma da un punto di vista di sviluppo è piuttosto negativo, poiché una volta compilati anche programmi molto semplici possono essere estremamente "grandi" in dimensioni fisiche (ovvero in kByte).

Il Turbo Pascal utilizza come libreria standard *Turbo.tpl*. Questa contiene tutte le funzioni di base. Non viene mai invocata, perché lo fa in automatico il compilatore durante la compilazione. Tuttavia, come abbiamo già visto, esistono molte altre librerie.

Dall'ultimo programma osservato si nota come deve essere fatta la dichiarazione delle librerie:

```
uses nomelibreria1, nomelibreria2, ...;
```

uses è una parola riservata, quindi non può essere usata come nome di variabile o funzione. La dichiarazione delle librerie deve precedere quella delle variabili.

Le librerie e le rispettive funzioni analizzate in questa pagina sono chiamate "**librerie predefinite**" in quanto sono di norma distribuite di default dal compliatore in uso. È infatti possibile anche creare librerie personalizzate per poter agevolare il proprio lavoro in più programmi differenti.

Crt

La libreria di gran lunga più usata è sicuramente *crt*, con tutte le funzioni relative all'estetica in ambito DOS. Ma non abbiamo ancora affrontato il problema dell'utilizzo di librerie all'interno dei nostri programmi.

Probabilmente avrete notato che eseguendo i programmi implementati finora l'esecuzione terminava istantaneamente senza l'aggiunga del `readln` finale, non appena il risultato veniva mostrato sullo schermo. Questo può essere abbastanza scomodo. Ma anche per questi problemi aiuta la libreria CRT, con la funzione `readkey()` che, come abbiamo già visto, permette di leggere il primo carattere che viene immesso da tastiera.

Funzioni di crt

```
clrscr;
```

Questa funzione permette di pulire lo schermo e di posizionare il cursore in alto a sinistra in posizione (1,1)

```
cursoroff;
cursoron;
```

Queste due funzioni permettono di spegnere o accendere il cursore.

```
gotoXY(posX, posY);
```

Questa procedura permette di posizionare il cursore in una posizione precisa sullo schermo.

```
sound(hz);
nosound;
```

La funzione *sound* fa emettere alla macchina un suono alla frequenza *hz*. Per fermare questo suono bisogna assolutamente utilizzare la procedura *nosound*

```
delay(time);
```

Questa funzione fa sì che il sistema si fermi in pausa per un certo numero di millisecondi definito da *time*. L'uso della procedura `delay` è tipicamente usata quando si fa uso di suoni. La struttura d'uso generalmente è la seguente:

```
sound(440);
delay(1000);
nosound;
```

Il risultato è un LA di durata un secondo. La funzione delay() è però basata sul clock del processore per calcolare il tempo, quando è stato creato il Pascal ovviamente i processori non avevano la potenza di oggi. La funzione *delay()* è quindi difficile da controllare ma se si vogliono ottenere ritardi definiti con bassa precisione cronometrica si possono inserire più delay() di seguito o sfruttare i cicli.

```
keypressed
```

Questa funzione è di tipo boolean, inizialmente è false e restituisce true appena l'utente preme un tasto. Viene spesso usata con il ciclo di tipo repeat... until:

```
repeat
   ...
   ...
until keypressed;
```

in questo modo le istruzioni vengono ripetute fino a quando l'utente non preme un tasto.

readkey

Funzione che permette di leggere un carattere inserito da tastiera senza visualizzarlo. Restituisce un valore di tipo char. Per recuperare il valore letto si utilizza la notazione:

```
x:=readkey;
```

Se si utilizza dopo "keypressed", si legge direttamente il carattere premuto per fare diventare "keypressed" false. Esempio:

```
repeat
  ...
  ...
until keypressed;
x:=readkey;
```

a questo punto l'utente ha premuto un tasto, con il quale è uscito dal ciclo e ha assegnato il carattere char alla variabile X.

textcolor(*numero*);
textbackground(*numero*);

Permettono di cambiare colore al testo e allo sfondo rispettivamente. Il codice di colore può anche essere il nome in inglese del colore. Questo perché nella libreria *crt* sono definite anche alcune costanti fra cui proprio quelle relative ai colori. Fra l'altro si può utilizzare anche la costante *blink*, per creare testo intermittente. Questo uso deve essere fatto così:

```
textcolor(white+blink);
```

Tuttavia, se per compilare il programma si usa Turbo Pascal, usando la unit *crt* quando viene compilato il programma, potrebbe comparire l'errore *Runtime Error 200: divide by 0* pur senza che vi sia alcuna divisione per 0 all'interno del programma. Questo è dovuto a un problema d'incompatibilità, che non si verifica però se si usa un altro compilatore.

Tabella: le frequenze delle note

Le frequenze delle note

Nota	Frequenza (Hz)	Formula (Hz)
Do	262	$440 \cdot (2^{\frac{1}{12}})^{-9}$
Do#/Reb	277	$440 \cdot (2^{\frac{1}{12}})^{-8}$
Re	294	$440 \cdot (2^{\frac{1}{12}})^{-7}$
Re#/Mib	311	$440 \cdot (2^{\frac{1}{12}})^{-6}$
Mi	330	$440 \cdot (2^{\frac{1}{12}})^{-5}$
Fa	349	$440 \cdot (2^{\frac{1}{12}})^{-4}$
Fa#/Solb	370	$440 \cdot (2^{\frac{1}{12}})^{-3}$
Sol	392	$440 \cdot (2^{\frac{1}{12}})^{-2}$
Sol#/Lab	415	$440 \cdot (2^{\frac{1}{12}})^{-1}$
La	**440**	$440 \cdot (2^{\frac{1}{12}})^{0}$

La#/Sib	466	$440 \cdot (2^{\frac{1}{12}})^1$
Si	494	$440 \cdot (2^{\frac{1}{12}})^2$

La frequenza di riferimento è quella del La, ovvero 440 Hz. Per ottenere le altre ottave basta raddoppiare o dimezzare le frequenze della nota corrispondente di questa ottava centrale.

Graph

graph è una libreria che fornisce la possibilità di integrare la grafica nelle applicazioni Pascal.

Funzioni di graph

Prima di iniziare a disegnare, è necessario entrare in modalità grafica tramite le funzioni:

```
scheda := detect;
initgraph(scheda, modo, percorso); es: initgraph(scheda,triangolo,'c:\FPC');
```

dove scheda e modo sono due variabili da dichiarare come integer nell'intestazione del programma. La funzione `detect` ricerca la scheda grafica in uso e restituisce il valore numerico ad essa associato; `initgraph` avvia la modalità grafica utilizzando la scheda identificata dal valore numerico di `scheda` e attribuisce alla variabile `modo` la modalità grafica selezionata automaticamente da Pascal; `percorso` è una stringa che identifica il percorso nel quale sono presenti i driver della scheda grafica: normalmente è sufficiente indicare il percorso di installazione del compilatore, ad esempio C:\TP per Turbo Pascal e C:\FPC per Free Pascal.

Quando la modalità grafica viene attivata, viene aperta una finestra aggiuntiva che sarà la schermata di output delle funzioni grafiche. Per chiuderla, usiamo la funzione

```
closegraph;
```

Per usare i colori abbiamo a disposizione le funzioni:

```
setcolor(colore) {imposta il colore di primo piano}
```

Per identificare un colore possiamo usare un numero da 0 a `getmaxcolor` (è una funzione che restituisce il numero massimo di colore disponibile) oppure le costanti red, white, black, ecc...

Un altro concetto importante è quello di puntatore attivo, cioè la posizione nel quale si trova il puntatore grafico. Quando vengono effettuati dei disegni di punti e linee, viene spostato il puntatore attivo in una posizione particolare.

Vediamo ora una serie di funzioni di disegno:

```
putpixel(x, y, colore)
```

Traccia un punto di coordinate x, y e del colore scelto e sposta in x, y il puntatore attivo.

```
getpixel(x,y)
```

Restituisce il colre del pixel in posizione x, y.

```
moveto(x,y)
```

Sposta il puntatore attivo nella posizione x,y.

```
GetX
GetY
```

Restituiscono le coordinate del puntatore attivo.

```
GetMaxX
GetMaxY
```

restituiscono i valori massimi della x e della y (ovvero le dimensioni in pixel della finestra di output).

```
line(x1, y1, x2, y2)
```

Traccia una linea dal punto x1, y1 al punto x2, y2. Sposta il puntatore attivo su x2, y2.

```
linerel(dx, dy)
```

Traccia una linea partendo dal puntatore attivo fino alla posizione x+dx e y+dy (dove x e y sono le coordinate del puntatore).

```
lineto(x, y)
```

Traccia una linea dal puntatore attivo al punto di coordinate x,y.

```
circle(x, y, r)
```

Disegna una circonferenza di centro x, y e di raggio r. Sposta il puntatore attivo su x, y.

```
 arc(x,y, ang1, ang2, r)
```

Disegna un arco di circonferenza di centro x, y e di raggio r a partire dall'angolo di gradi ang1 fino ad ang2. I gradi sono contati in senso antiorario a partire dalle ore 3. Sposta il puntatore attivo su x, y.

```
ellipse(x,y,ang1, ang2, rx, ry)
```

Disegna un arco di ellise di centro x, y tra gli angoli ang1 e ang2 (per una ellisse completa usare i valori 0 e 359) con il raggio orizzontale rx e raggio verticale ry. Sposta il puntatore attivo su x, y.

Graph3

Un'altra libreria molto usata è *graph3* che permette l'utilizzo della grafica delle versione di Turbo Pascal 3.xx, e tramite semplici comandi si riesce a disegnare facilmente punti, linee e altre forme geometriche. Questa libreria fornisce ovviamente meno funzionalità della sua pronipote *graph*, ma non richiede l'uso di una scheda grafica. L'utilizzo di questa libreria sostituisce alcune funzioni della libreria *crt* come *clrscr*, che viene sotituito da *clearscreen*. Dal momento in cui si entra nella modalità grafica con il comando *graphcolormode* la grandezza dei caratteri aumenta e la risoluzione diminuisce.

Funzioni di graph3

Per attivare la modalità grafica è possibile usare due funzioni:

```
graphmode
```

permette di passare in modalità grafica in bianco e nero;

```
graphcolormode
```

è invece a colori. Dopo aver selezionato la modalità grafica a colori, è possibile scegliere una delle tavolozze di colori disponibili tramite la procedura

```
palette(n)
```

dove n è un intero compreso tra 0 e 3. A seconda del valore di *n*, il colore rappresentato dai diversi numeri cambia:

Tavolozza	Sfondo	Numero colore		
		1	2	3
0	nero	verde	rosso	marrone
1	nero	turchese	magenta	grigio chiaro
2	nero	verde chiaro	rosso chiaro	giallo
3	nero	turchese chiaro	magenta chiaro	bianco

```
plot(x,y,c)
```

Questa funzione permette di disegnare un punto di coordinate *x*, *y* di colore *c* che può che rappresenta un colore diverso a seconda della palette usata (vedi la tabella).

```
draw(x1,y1,x2,y2,c)
```

Questa funzione permette di disegnare una linea dal punto *x1*, *y1* al punto *x2*, *y2* di colore *c*

Istruzioni di controllo

Strutture di selezione

Selezione binaria

In alcuni degli esempi precedenti si è analizzata la struttura *if...then...else*; vediamo nello specifico in cosa consiste.

Nella pratica, questo costrutto consente la scelta fra due alternative: **se** (*if*) una certa condizione è vera **allora** (*then*) il programma esegue una certa istruzione (semplice o composta), **altrimenti** (*else*) ne esegue un'altra:

```
if condizione  then istruzione
       else istruzione;
```

che come abbiamo già visto permette la scelta fra un caso vero e un caso falso e in base a questo esegue il contenuto del *then* o dell'*else*; tuttavia a volte il ramo in *else* non é; strettamente utile o addirittura non serve.

Quando le istruzioni che seguono il *then* o l'*else* sono più di una esse devono essere delimitate da istruzioni *begin* e *end* (sono istruzioni composte):

```
if condizione then
begin
    istruzione 1;
    istruzione 2;
    ...
    istruzione n
end
else
begin
    istruzione 1;
    istruzione 2;
    ...
    istruzione n
end;
```

Si noti come le istruzioni che precedono *end* ed *else* non debbano essere seguite da punto e virgola

Facciamo un esempio pratico.
Si vuole calcolare la radice quadrata di un numero: ciò è possibile nell'insieme dei numeri reali solo se il valore in ingresso è positivo. Il programma deve perciò:

- acquisire il numero
- eseguire una selezione binaria: *se* il numero è positivo,*allora* ne calcola la radice quadrata e espone il risultato;*altrimenti* stampa un messaggio di errore;

```
program radice_quadrata;
    var n,r:real;
begin
    writeln('Inserisci un numero positivo');
    readln(n);
    if n>=0 then
    begin
       r:=sqrt(n);
       writeln('La radice quadrata del numero inserito e'' ',r:8:3);
    end
    else
       writeln('La radice di un numero negativo non può essere espressa
 con numeri reali');
    readln;  (* attenzione: questo readln non fa parte del blocco else!
*)
end.
```

Selezione multipla

Con l'uso del costrutto `case... of` è possibile eseguire selezioni più complesse. La sintassi è:

```
case espressione of
     valore 1 : begin
                    istruzioni;
                end;
     valore 2 : begin
                    istruzioni;
                end;
     ...

     valore n : begin
                    istruzioni
                 end;

     else        begin
                    istruzioni;
                 end;
end;
```

Con il costrutto *case*, il valore dell'espressione in testa viene confrontato con il valore di ogni singolo caso; quando viene trovato un valore che soddisfa le condizioni sul selettore vengono eseguite le istruzioni relative al blocco in

questione; poi il controllo passa alla prima istruzione dopo il costrutto *case*. Se nessun valore soddisfa il selettore viene eseguita l'ultimo blocco di istruzioni individuato dalla parola chiave **else** (se questo blocco è stato inserito). Se la parte con *else* viene omessa e il selettore non eguaglia nessuno dei casi in elenco, si passa anche in questo caso alla prima istruzione dopo il costrutto *case*.

Quando c'è solo un'istruzione che segue un certo caso si possono omettere *begin* e *end* (questo perché, in realtà, le istruzioni delimitate da begin ed end si possono considerare come un'unica istruzione).

Occorre fare attenzione che il costrutto *case* termina con un *end*. Facciamo un esempio del costrutto case:

```
case contatore of
  1, 2, 50, 210: writeln(contatore);
  30, 60..80 : begin
                 writeln(variabile);
               end;
end;
```

Ecco di seguito alcune considerazioni sull'esempio proposto:

- è stata omesso il blocco facoltativo **else**
- sono state indicate delle liste di possibili casi
- è stato usato un range di valori x..y; al terzo rigo,*60..80* indica tutti i valori compresi tra 60 e 80 (inclusi).

Strutture Iterative

In Pascal abbiamo diversi tipi di strutture iterative (in genere vengono detti "*cicli*"). Una struttura iterativa prevede la ripetizione di un'istruzione semplice o composta secondo delle condizioni poste all'inizio o alla fine della struttura.

Iterazione con contatore

Questa struttura ci permette di ripetere una determinata serie di istruzioni per un numero finito di volte modificando man mano il valore di una variabile. La sua sintassi è:

```
for Contatore:=valore_iniziale to valore_finale do
begin
  istruzione 1;
  istruzione 2;
  ...
  istruzione n;
end;
```

Nella pratica questo costrutto consente la ripetizione di un certo gruppo di istruzioni fino a quando la variabile contatore raggiunge un certo valore: **per** (*for*) contatore:=valore_iniziale **fino a** (*to*) valore_finale **esegui** (*do*) un blocco d'istruzioni racchiuso tra *begin* ed *end*.

`Contatore` è una variabile (di solito vengono usate le lettere I,L,M,N,J,K) che da un valore iniziale cresce di un'unità ad ogni ripetizione fino a che non raggiunge il valore finale, che rappresenta il valore per il quale il ciclo terminerà.

L'iterazione può anche procedere in modo che la variabile contatore decresca a ogni ciclo; in questo caso la sintassi è:

```
for Contatore:=valore_iniziale downto valore_finale do
begin
  istruzione 1;
```

```
  istruzione 2;
  ...
  istruzione n;
end;
```

Facciamo un esempio: implementiamo un programma che calcoli la somma dei primi n numeri naturali usando un ciclo `for` (senza usare la nota formula $\frac{n(n+1)}{2}$).

Il programma deve:

- leggere n;
- tramite un ciclo for, sommare tutti i numeri da 1 a n;
- esporre il risultato.

```
program sommatoria;
    var i,n,s:integer;
begin
    s:=0;
    writeln('Inserisci n');
    readln(n);
    for i:=1 to n do
        s:=s+i;
    writeln('La somma dei primi n è ',s);
    readln;
end.
```

Analizziamo il programma riga per riga:

- dichiarazione del programma;
- dichiarazione delle variabili: i è la variabile contatore, n funge da valore finale del ciclo for, q è di volta in volta il quadrato di i mentre s è la somma dei numeri naturali fino a n;
- inizio del programma
- la somma è inizialmente uguale a 0;
- messaggio di testo;
- il programma legge n;
- il ciclo for: *per i uguale a 1 fino a che i non sia uguale a n,esegue il gruppo d'istruzioni:*
- inizio del ciclo;
- alla variabile s viene assegnata la somma del valore di s+i; prima del ciclo s era uguale a 0, quindi s:=1;
- fine del ciclo: se i è diverso da n, il ciclo viene riavviato; altrimenti si prosegue all'istruzione successiva. Automaticamente alla fine del ciclo i viene incrementato di 1.

Vediamo cosa succede se n assume ad esempio il valore 50:

- i := 2, diverso da 50, quindi il ciclo continua;
- s := s + i, cioè s:=1+2; infatti la variabile s aveva valore 1 alla fine della prima fase del ciclo.
- ricomincia il ciclo; i:=3,diverso da 50,quindi il ciclo continua;
- s := s + i;cioè s := 3 + 3;
- e così via, fino a quando i sia diverso da 50.

Iterazione con condizione in testa

Una struttura di iterazione con condizione in testa è tradotta in Pascal con il ciclo **while... do** La sua sintassi è:

```
while condizione do
begin
    istruzione 1;
    istruzione 2;
    ...
    istruzione n;
end;
```

Ovvero: **mentre** (*while*) una determinata condizione è vera **esegui** (*do*) le istruzioni all'interno del ciclo.

Fino a quando la condizione è vera, il ciclo viene eseguito. Tra **begin** ed **end** vanno inserite le istruzioni che cambiano in qualche modo il valore di verità della condizione. Questo è molto importante perché altrimenti il ciclo si ripeterebbe all'infinito e il programma non riuscirebbe mai a giungere alle righe che seguono l'istruzione **end**. Questo è un comune errore che con un po' di attenzione può essere evitato. Se vi è una sola istruzione **begin** e **end** possono essere omessi. Ad esempio:

```
...
x:=1;
while x<>5 do
    x:=x+1;
...
```

In questo esempio viene eseguito quanto indicato all'interno del ciclo finché viene verificata la condizione, cioè ad x viene sommato 1 fino a che il suo valore non sia uguale a 5,e quindi la condizione iniziale del ciclo diventi falsa.

Iterazione con condizione finale

Per chiudere il discorso riguardo alle strutture di controllo analizziamo il ciclo con condizione finale, che in Pascal si traduce con un ciclo **repeat... until**

La sua sintassi e':

```

repeat
begin
    istruzione 1;
    istruzione 2;
    ...
    istruzione n
end;
until condizione;
```

In sostanza questo ciclo ripete le istruzioni comprese tra *Repeat* ed *Until* (dall'inglese,*finché*) fino al verificarsi della condizione espressa, detta anche condizione di uscita. Anche in questo caso occorre porre attenzione al fatto che la condizione di uscita diventi vera in qualche modo, altrimenti finiamo in un loop infinito perdendo il controllo del programma.

Ad esempio:

```
x:=1;
repeat
```

```
x:=x+1;
until x=5;
```

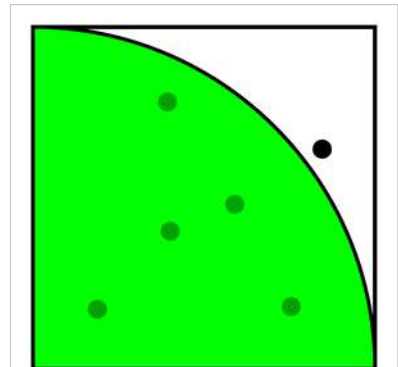
In questo caso x viene sommato a 1 fino a che il suo valore non sia diverso da 5; quando x=5 si esce dal ciclo.

Un confronto

Nel caso del ciclo **For** il numero di cicli eseguiti è noto, poiché il contatore parte da un elemento iniziale ed arriva fino all'elemento finale. Nel caso di **While** e di **Repeat Until** il numero di volte che viene ripetuto il ciclo generalmente non è noto a priori, in quanto dipende dal cambiamento che subisce la variabile che controlla la condizione.

È importante notare che le istruzioni del ciclo **Repeat Until** verranno eseguite almeno una volta poiché la condizione viene verificata alla fine, dopo il codice del ciclo; un ciclo **While**, essendo la sua condizione testata prima dell'esecuzione del codice associato al ciclo, potrebbe, se questa risulta falsa, non fare nulla e riprendere dalle istruzioni che si trovano dopo il ciclo. Questo può risultare utile in particolari casi come in altri può essere causa di bug.

Esercizi

- Scrivere un programma che stampi la tabellina del numero immesso dall'utente
- Scrivere un programma "indovina il numero". Il programma deve chiedere all'utente un numero, scegliere casualmente un numero tra 1 e il numero fornito. L'utente dovrà quindi indovinare il numero pensato dal computer: ad ogni tentativo, il computer deve dire se il tentativo è maggiore o minore del numero pensato.
- Scrivere un programma per il calcolo di π usando il *metodo Monte Carlo*.
  Questo metodo consiste nell'estrarre casualmente dei punti all'interno di un quadrato e valutare se questi sono interni od esterni all'arco di ampiezza $\frac{\pi}{4}$ inscritto nel quadrato.
  Consigli: considerare un quadrato di raggio unitario; dato il punto di coordinate *x* e *y*, questo si trova dentro al cerchi se vale la relazione $\sqrt[2]{x^2 + y^2} < 1$.

Rappresentazione grafica del metodo Monte Carlo applicato al calcolo di π

Per approfondire vai su Wikipedia, vedi la voce ***Metodo Monte Carlo****.*

- Scrivere un programma per determinare il valore di un radicale tramite tentavi successivi, in questo modo: il programma chiede con quale indice si vuole lavorare; successivamente, chiederà all'utente un'approssimazione della radice, e mostrerà la potenza del numero immesso secondo l'indice. In questo modo, in base al valore della potenza, l'utente potrà ritentare proponendo un numero maggiore o minore di quello immesso. Il programma termina quando l'utente immette 0.

Esempio: si supponga si voglia trovare con tentativi successivi il valore della radice quadrata di 5. Si sceglierà innanzitutto di lavorare con indice 2, quindi si immette il primo tentativo (ad esempio 2). Il programma stamperà 4, che è troppo basso: serve un numero più alto. Immettendo 3 si ottiene 9, che è troppo alto: la radice del numero cercato sarà quindi compresa tra 2 e 3; immettiamo ad esempio 2.5. Il risultato è ancora troppo alto, immettiamo quindi 2.3: otteniamo 5.29, ancora troppo alto. Immettendo 2.2 otteniamo invece un numero più basso di due. Il

numero cercato è quindi compreso tra 2.2 e 2.3, e così via...

- Scrivere una funziona che restituisca la posizione di un carattere chiesto all'utente nella stringa chiesta anch'essa all'utente. La posizione deve essere quella della prima occorrenza trovata.

Array

Analizzeremo in questa unità l'uso degli **array**, un tipo di dato molto potente che consente di immagazzinare informazioni in modo semplice e veloce.

Vettori

Un metodo molto diffuso e comodo di rappresentazione e ordinamento di dati consiste nell'uso di una tabella.
Ad esempio:

Indice	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Valori	valore 1	valore 2	valore 3	valore 4	valore 5	valore 6	valore 7	valore 8	valore 9	valore 10

Se diamo un nome alla tabella, ad esempio *Array*, possiamo indicare il valore dell'elemento di indice 8 in questo modo:

```
Array[8] {indica l'elemento di posizione 8}
```

così come

```
Array[9] {indica l'elemento di posizione 9}
```

e così via.

Una struttura come quella dell'esempio precedente è chiamata **vettore**: a ogni posizione (1, 2, ..., 10) nella tabella corrisponde un valore (valore 1, valore 2, ..., valore 10): la prima colonna della tabella in questo caso si chiama *indice* del vettore.
Il vettore consente di identificare un gruppo di dati con un nome collettivo (*Array* dell'esempio precedente) e di poter individuare ogni singolo dato attraverso un indice racchiuso tra parentesi quadre, il quale rappresenta la posizione occupata dal dato nella tabella.

Anche in Pascal, così in altri linguaggi di programmazione, si possono strutturare i dati in maniera simile tramite gli **array**.

Gli array in Pascal

Per dichiarare un vettore in Pascal è necessario indicare le diverse posizioni possibili (normalmente si usano i numeri) e il tipo di dato che conterrà ciascun valore dell'array (nella nostra tabella, il tipo di dato dei valori della riga **Valore**):

```
var
 nome_variabile: array[1..n] of tipo_del_dato;
```

che indica che il vettore nome_variabile avrà come posizioni i numeri da 1 a n (la notazione *x..y* dove x e y sono due valori dello stesso tipo indica tutti i valori compresi tra *x* e *y*).
Notare che quello indicato tra parentesi non è altro che un *subrange*, e quindi sono ammesse anche dichiarazioni come:

```
var
scheda: array['a'..'z'] of integer
```

e così vale per gli altri tipi di dati, anche quelli enumerativi. Valgono anche dichiarazioni come:

```
type
    mese : (gen, feb, mar, apr, mag, giu, lug, ago, set, ott, nov, dic);
var
    nomi_mesi: array[mese] of string;
```

Per accedere ai valori dell'array, è sufficiente comportarsi come con le variabili normale, usando la notazione

```
nome_array[indice]
```

Ad esempio:

```
scheda['a'] := 45;
writeln(scheda['a']);
```

È possibile dichiarare gli array anche con un altro metodo:

```
type
  nome_del_tipo=array[1 .. n] of tipo_del_dato;
 var
  nome variabile:nome_del_tipo;
```

Per leggere e scrivere gli elementi di un vettore di N elementi in Pascal è possibile usare un ciclo for:

```
for I:=1 to N do
 readln(vettore[I]);

for I:=1 to N do
 writeln(vettore[I]);
```

All'aumentare del contatore I, il programma scorre i posti del vettore. Nel caso di array che utilizzano indici non interi, si userà ad esempio:

```
for I := 'a' to 'z' do
 readln(vettore[I]);
```

Array bidimensionali

Prendiamo come esempio questa tabella:

	1	2	3	4	5	6	7	8	9	10
1	array[1,1]	array[1,2]	array[1,3]	array[1,4]	array[1,5]	array[1,6]	array[1,7]	array[1,8]	array[1,9]	array[1,10]
2	array[2,1]	array[2,2]	array[2,3]	array[2,4]	array[2,5]	array[2,6]	array[2,7]	array[2,8]	array[2,9]	array[2,10]
3	array[3,1]	array[3,2]	array[3,3]	array[3,4]	array[3,5]	array[3,6]	array[3,7]	array[3,8]	array[3,9]	array[3,10]

Nel vettore, il posto nella tabella è individuato da un indice; nella tabella bidimensionale, invece, da una coppia di indici. Una tabella bidimensionale è anche chiamata **matrice**.

Una vettore bidimensionale si dichiara in questo modo:

```
var
 nome_variabile: array[1 .. n,1 .. m] of tipo_del_dato;
```

oppure, analogamente all'esempio precedente

```
type
 nome_del_tipo = array[1 .. n, 1 .. m] of tipo_del_dato;
var
 nome variabile: nome_del tipo;
```

Per leggere e scrivere i dati di una matrice di N righe e M colonne in Pascal si possono usare due cicli for annidati:

```
for I:=1 to N do
 for J:=1 to M do
  readln(array[I,J]);

for I:=1 to N do
 for J:=1 to M do
  writeln(array[I,J]);
```

Utilizzi

L'utilizzo degli array è molto comodo, perché permette di gestire facilmente grandi quantità di dati. Vediamo ad esempio questo programma per un ipotetico sistema di gestione degli alunni:

```
program verifica;
var numalunni, puntimax, i : integer;
    punti, voto: array[1..100] of real;
    sommavoti : real;
    op : char;
begin
    write('Quanti alunni? '); readln(numalunni);
    write('Qual era il punteggio massimo? '); readln(puntimax);
    for i := 1 to numalunni do
    begin
        write('Inserisci il punteggio per l''alunno ', i, ' ');
        readln(punti[i]);
    end;
    writeln('I dati sono stati memorizzati');
    {qui si possono effettuare altre operazioni}
    {qui si possono riprendere i dati memorizzati in precedenza}
```

```
    sommavoti := 0;
    for i := 1 to numalunni do
    begin
        voto[i] := (punti[i] / puntimax ) * 10;
        writeln('Il voto per l''alunno ', i, ' e'' ', voto[i]:3:1);
        sommavoti := sommavoti + voto[i];
    end;
    readln;
    writeln('La media dei voti e'' ', sommavoti/numalunni:3:1);
    readln;
end.
```

Un'altra utilità degli array è quella di poter accedere ad un valore anche senza saperne il suo nome, ma solo tramite un valore immesso dall'utente. Ad esempio, volendo convertire il numero di un mese nel nome corrispondente, senza array, bisognerebbe fare ricorso ad un costrutto *case... of*. Usando i vettori, invece, è sufficiente dichiarare un array `nomeMese[1..12]` e assegnare ad ogni posizione il nome del mese interessato. Per ottenere poi il nome del mese *n*, basterà scrivere `nomeMese[n]`.

Esercizi

- Creare un programma che stampi i primi *n* numeri della *successione di Fibonacci* (la serie di Fibonacci è una serie di numeri F dove $F_0 = 0$, dove $F_1 = 1$ e ciascun altro dove $F_n = F_{n-1} + F_{n-2}$)

Metodo top-down, procedure e funzioni

Per **metodo top-down** si intende una suddivisione di un problema, di un algoritmo o di un procedimento in sottoproblemi più piccoli e più semplici da implementare nel linguaggio desiderato.
Una delle comodità della scomposizione dei problemi in porzioni di codice è la loro riutilizzabilità: tramite il metodo top-down, infatti, il programmatore può definire blocchi di codice a cui è possibile fare riferimento durante il corso della programmazione.

In Pascal possiamo implementare soluzioni top-down tramite le **procedure** e le **funzioni**.

Procedure

Per procedura si intende una porzione di codice riutilizzabile che può prevedere parametri in ingresso ma non prevede parametri in uscita. La sua sintassi è:

```
procedure nome_procedura(variabili_richieste_come_parametri);
dichiarazioni
begin
istruzioni;
end;
```

Le diverse procedure del programma Pascal devono essere scritte prima del blocco del blocco begin...end che delimita il programma principale. Vediamo un esempio completo di un sottoproblema implementato in Pascal come, ad esempio, dare il benvenuto ad un utente, all'interno di un programma più lungo che ometteremo:

```
program Esempio;
```

```
    var [...]


    procedure Benvenuto;
        var nome:string[50];
    begin
        writeln('Come ti chiami?');
        readln(nome);
        writeln('Benvenuto su it.wikibooks, ', nome);
    end;


(* qui incomincia il programma vero *)
begin
    (* centinaia righe di codice... *)
    (* in questa riga viene ''chiamata'' (o ''invocata'') la procedura
Benvenuto *)
    (* e viene eseguita la porzione di codice in essa contenuta *)
    Benvenuto;
    (* centinaia righe di codice... *)
end.
```

In questo semplice caso la procedura Benvenuto chiede un input all'utente e stampa un messaggio di benvenuto.
La comodità di questo semplice spezzone sta nel fatto che, durante l'esecuzione del programma vero e proprio, in qualsiasi punto è possibile eseguire la procedura Benvenuto quante volte si vuole riducendo così la mole di codice da scrivere e anche facilitando, ad esempio, la correzione di eventuali errori o la revisione del codice.
Questo metodo è ovviamente comodo nel caso di programmi molto lunghi o di sottoproblemi che si ripresentino molte volte nel corso del programma

Le variabili usate nella procedura e dichiarate quindi nella sezione di dichiarazione della procedura stessa (nel nostro esempio la variabile nome) sono chiamate variabili **locali**, in quanto non è possibile richiamarle se non dalla procedura nelle quali sono dichiarate. Nel programma vero e proprio e nelle procedure le variabili locali delle altre eventuali procedure non sono quindi utilizzabili.
Sono chiamate varibili **globali** le variabili dichiarate all'intestazione del programma principale, in quanto possono essere richiamate e utilizzate sia in ambito del programma principale stesso sia in ambito locale delle procedure.
Si noti che nel caso che due variabili con lo stesso nome sono dichiarate sia in ambito globale che in ambito locale, nelle procedure il compilatore prende in considerazione la variabile locale. Il consiglio è comunque quello di dare nomi sempre diversi alle variabili, evitando sovrapposizioni di qualsiasi genere.
Un esempio potrebbe rendere il tutto più chiaro:

```
program Variabili;

    var principale:integer;


    procedure proc1;
        var locale1:integer;
    begin
        (* da qui sono accessibili solo le variabili ''locale1'' e
```

```
''principale''*)
    end;


    procedure proc2;
        var locale2:integer;
    begin
        (* da qui sono accessibili solo le variabili ''locale2'' e
''principale''*)
    end;


(* qui incomincia il programma vero *)
begin
    (* da qui è accessibile solo la variabile ''principale''*)
end.
```

Procedure con argomenti

Può risultare utile, in molti casi, passare alla procedura dei valori che specifichino meglio l'operazione da svolgere o che ne rendano l'uso più utile alle diverse situazioni. Questi valori sono detti **argomenti** o **parametri** Nell'esempio precedente, ad esempio, sarebbe meglio poter specificare ogni volta il messaggio stampato dalla procedura Benvenuto in modo tale da renderla utilizzabile in più situazioni. Vediamo l'esempio con l'uso delle variabili:

```
program Esempio;

    var [...]


    procedure Benvenuto (domanda, risposta: string[100]);
        var nome:string[50];
    begin
        writeln(domanda);
        readln(nome);
        writeln(risposta, nome);
    end;


(* qui incomincia il programma vero *)
begin
    (* centinaia righe di codice... *)
    Benvenuto('Qual è il tuo nome?', 'Benvenuto nel mio sito, ');
    (* centinaia righe di codice... *)
    Benvenuto('Come si chiama la tua fidanzata?', 'Salutami allora ');
end.
```

In questo caso la procedura Benvenuto è stata chiamata due volte nel corso del programma *passando* ogni volta i due argomenti richiesti, che non sono altro che delle espressioni che verranno poi salvate nelle variabili *domanda* e *risposta*. L'output delle due procedure sarà il seguente, nel caso l'input sia Luigi o Luisa:

```
Qual è il tuo nome? Luigi
Benvenuto nel mio sito, Luigi
```

```
Come si chiama la tua fidanzata? Luisa
Salutami allora Luisa
```

I parametri sono ovviamente variabili locali della procedura.

Passaggio di parametri per valore e per riferimento

Introdotto l'uso dei parametri, è necessario però fare una distinzione molto importante tra i due modi possibili di passare una variabile:

- quando i valori sono passati per **valore** la variabile che funge da parametro assume semplicemente il valore dell'espressione introdotta nella chiamata della procedura. Questa è la situazione presentata nel programma Esempio che abbiamo precedentemente visto.
- quando i valori sono passati per **riferimento** la variabile che funge da parametro si sostituisce momentaneamente alla variabile passata nella chiamata della procedura, che verrà quindi modificata nel corso del programma. In questo caso il parametro deve essere indicato con la sintassi `var nome_var : tipo di dato;`

La differenza sarà forse più chiara con un esempio:

```
program Esempio;

    var z1, z2:real;


    procedure Valore (x:real);
    begin
        x := 2*x;
        writeln('Il valore di X è ', x);
    end;


    procedure Riferimento (var x:real);
    begin
        x := 2*x;
        writeln('Il valore di X è ', x);
    end;


(* qui incomincia il programma vero *)
begin
    z1 := 5;
    z2 := 12;
    Valore(z1);
    Riferimento(z2);
    writeln('Il valore di z1 è ',z1);
    writeln('Il valore di z2 è ',z2);
end.
```

Dopo l'esecuzione del programma, avremo quindi la seguente situazione:

- il valore di z1 sarà rimasto lo stesso di quando la procedura è stata invocata, in quanto il passaggio del parametro x è avvenuto per valore. Al posto di z1 si poteva passare alla procedura anche un'espressione, come `3 + 4` o anche `z1 + 5`.
- il valore di z2 dopo la chiamata della procedura Riferimento sarà pari a 24, ossia 12 * 2, in quanto il parametro x è stato passato oper riferimento e, quindi, al momento dell'istruzione `x := x*2` non varia solo la variabile x stessa all'interno della procedura ma anche il valore della variabile z2.

L'output del programma sarà quindi:

```
Il valore di X è 10
Il valore di X è 24
Il valore di z1 è 5
Il valore di z2 è 24
```

Funzioni

Il concetto di funzioni in programmazione è strettamente legato al concetto di funzione matematica. In Pascal possiamo pensare ad una funzione come ad una procedura che restituisce un valore: le funzioni, come le procedure, devono essere dichiarate prima del programma principale e possono disporre di variabili locali e di parametri.

La loro sintassi è tuttavia leggermente diversa:

```
function nome_della_funzione(parametri):tipo_di_dato_restituito_dalla_funzione;
dichiarazioni
begin
istruzioni;
end
```

Per riferirsi al valore della funzione si deve fare riferimento al nome della funzione stessa. Creiamo ad esempio una funzione che restituisca il valore assoluto di un numero:

```
function Assoluto (x:real):real;
begin
    if x<0 then
     Assoluto := -x
    else
     Assoluto := x;
end;
```

Analizziamo il listato riga per riga:

1. la prima riga è la dichiarazione della funzione, che richiede un parametro real, *x*, e che restituisce un valore real
2. inizio della funzione (non ci sono variabili da dichiarare in questo caso perché la funzione è molto semplice)
3. se x è minore di zero allora
4. restituisci come valore l'opposto di x
5. altrimenti (x maggiore o uguale a 0)
6. restituisci il valore di x
7. fine della funzione

In questo modo, passando alla funzione 3, Assoluto restituisce 3, mentre se si passa -12 la funzione restituisce 12. È possibile in qualsiasi punto del programma in cui la funzione è stata inserita ricorrere ad essa usando la seguente notazione:

```
Assoluto(n);
```

che funge da espressione in quanto restituisce un valore.

Ad esempio un semplice programma che utilizzi la funzione Assoluto potrebbe essere scritto così:

```
program Esempio;
var n: real;

function Assoluto (x:real):real;
begin
    if x<0 then
     Assoluto := -x
    else
     Assoluto := x;
end;

begin
    write('Calcola il valore assoluto del numero ');
    readln(n);

    writeln('Il valore assoluto di ', n:10:3, ' vale ',
Assoluto(n):10:3 );
end.
```

Ricorsività

È possibile anche prevedere che una funzione richiami se stessa: in questo caso si parla di **ricorsività**.

Un esempio tipico di funzione ricorsiva è quella di fattoriale. Il fattoriale di un numero $n \in \mathbb{N}$ (si indica con $n!$) è uguale al prodotto di $n \cdot (n-1) \cdot (n-2) \cdot (n-3) \ldots \cdot 1$ Eccone l'implementazione in Pascal:

```
function fattoriale (n: integer): longint;
begin
    if (n = 0) or (n = 1) then
        fattoriale := 1;
    else
        fattoriale := n * fattoriale(n-1);
end;
```

Analizziamo il caso di $n = 3$:

- viene restituito il valore $3 \cdot 2!$
- a sua volta $2!$, questo restituisce $2 \cdot 1!$
- il risultato di $1!$ è 1, quindi finisce la ricorsività
- a questo punto, avremmo che il risultato sarà $3 \cdot 2 \cdot 1$, che è quello che volevamo ottenere

Esercizi

- Scrivere una funzione *ipotenusa* che, presi come valori i cateti di un triangolo rettangolo, restituisca il valore della sua ipotenusa, applicando il teorema di Pitagora
- Implementare una funzione che calcoli l'*n*esimo numero della successione di Fibonacci, già vista nel capitolo precedente, utilizzando però un algoritmo ricorsivo.
- Implementare una funzione `strpos` che prenda come parametri una stringa e un carattere, e restituisca la posizione della prima occorrenza del carattere nella stringa. Se il carattere non viene trovato, restituisce -1.
- Una volta risolto il precedente esercizio, non dovrebbe essere difficile scrivere una funzione simile, ma che parta dal fondo a cercare il carattere
- Implementare una funzione `substr` che prenda come parametri una stringa e due interi, e restituisca una sottostringa partendo dal carattere in posizione specificata per una lunghezza specificata. Ad esempio se si passano come valori `'il linguaggio pascal'`, `4` e `5`, restituisce `'lingu'`.

Programmazione ad oggetti

Programmazione ad oggetti

La **programmazione orientata agli oggetti** (OOP) è un paradigma di programmazione che consente di gestire applicazioni complesse, come applicazioni molto grandi o con interfacce grafiche elaborate, con semplicità.

Quest'obbiettivo si raggiunge raggruppando in *classi* le strutture dati e le procedure che le elaborano ottenendo appunto degli *oggetti* caratterizzati da proprietà (i dati) e metodi (le procedure e le funzioni).

Il vecchio paradigma di programmazione, quella strutturale (o metodo *top-down*) poco si prestava all'elaborazione di programmi complessi. Le nuove architetture hardware erano però in grado di gestire programmi più potenti, che rendevano obsoleta di fatto la programmazione strutturale. Fu così che nacque la programmazione agli oggetti: non è più il programma in sé a gestire l'intero flusso di istruzioni, ma sono gli oggetti che interagiscono fra di loro intervenendo opportunamente sull'andamento del programma, specialmente se si utilizza il concetto di "evento" di cui ne parleremo quando avremo a che fare con Turbo Vision.

La programmazione orientata agli oggetti è pertanto un'estensione dell'originale linguaggio Pascal, ed è sopportata solo dai moderni compilatori come Free Pascal o Delphi, che consentono all'utente di creare applicazioni con classi e oggetti.

La prima versione di Turbo Pascal con l'OOP è stata il Turbo Pascal 5.5, e la versione 6.0 includeva il framework Turbo Vision, ovvero una vasta libreria che consente la creazione di una TUI (Text User Interface) adeguata. La IDE del Free Pascal è stat sviluppata con una libreria analoga, ma open source: la Free Vision.

Anche se il Pascal supporta ormai da tempo la programmazione orientata agli oggetti e la programmazione avanzata in generale, la mancanza di compilatori versatili che funzionino in ambiente Windows ha relegato il Pascal ad un ambito esclusivamente didattico.

Di seguito un semplice esempio di programma che implementa la programmazione a oggetti.

Esempio

Il programma memorizza un numero (1) nella variabile privata One, lo incrementa e lo stampa.

```
program types;   // Piccolo esempio OOP

type MyType=class
      private
       One:Integer;
      public
       function Myget():integer;
       procedure Myset(val:integer);
       procedure Increase();
     end;
function MyType.Myget():integer;
begin
  Myget:=One;
end;
procedure MyType.Myset(val:integer);
```

```
begin
  One:=val;
end;
procedure MyType.Increase();
begin
  One:=One+1;
end;

var
  NumberClass:MyType;
begin
  NumberClass:=MyType.Create;   // crea l'istanza dell'oggetto
  NumberClass.Myset(1);
  NumberClass.Increase();
  writeln('Result: ',NumberClass.Myget());
  NumberClass.Free;   // distrugge l'istanza
end.
```

Il programma mostrato è banale e non ha alcuno scopo se non quello di mostrare un esempio di sintassi OOP. Esempi molto più complessi si trovano sul sito del progetto FreePascal [1] e di Lazarus [2], un RAD per lo sviluppo di applicazioni OOP in FreePascal.

Note

[1] http://www.freepascal.org/
[2] http://lazarus.freepascal.org/

Concetti fondamentali

La programmazione orientata agli oggetti gira attorno a tre concetti fondamentali e di cruciale importanza. Questi concetti sono:

- L'incapsulamento
- L'ereditarietà
- Il polimorfismo

L'incapsulamento

Per un buon approccio alla programmazione agli oggetti, la prima cosa che si deve fare è stabilire quali sono gli oggetti, cosa fanno, quali sono le loro caratteristiche e come interagiscono fra di loro.

Abbiamo detto che gli oggetti sono strutture che contengono **campi** e **metodi**. Per campi si intendono delle variabili (in senso molto lato) o, meglio, dei *puntatori* a zone di memoria. I metodi invece sono le procedure, le funzioni, i costruttori e i distruttori (concetti che vedremo più avanti) legati al particolare oggetto.

I metodi sono pertanto la parte attiva di un oggetto, che possono modificare le caratteristiche dello stesso, ovvero possono modificare i campi. Il procedimento con il quale si definiscono i metodi e i campi di un oggetto è chiamato **incapsulamento**. Per essere più precisi, a venire incapsulata è la **classe**, che sarebbe un "tipo", un "modello" di oggetto. Sebbene nella sintassi del Turbo Pascal venga utilizzata la parola chiave `object` per definire una classe, ciò non deve creare confusione.

Un oggetto per essere creato ha bisogno di essere *istanziato* da una classe; quindi per esempio avendo una classe `Studente`, da questa classe possono venire istanziati due oggetti-*studente*: Francesco e Giorgio.

Qui di seguito un esempio in codice per creare una classe:

```
TYPE TStudente = OBJECT
                    Nome,Cognome: String;
                    AnnoDiNascita: integer;
                    FUNCTION GetNome : String;
                    FUNCTION GetCognome : String;
                    FUNCTION GetAnnoNascita : integer;
                    PROCEDURE SetNome(nome:string);
                    PROCEDURE SetCognome(cognome:string);
                    PROCEDURE SetAnnoNascita(anno:integer);
                 END;
```

In questo modo abbiamo definito la classe TStudente che contiene i campi Nome, Cognome e AnnoDiNascita, e i metodi per memorizzare i dati o per ottenerli dall'esterno.

L'ereditarietà

La programmazione orientata agli oggetti utilizza per gli oggetti una gerarchia ben definita che si rivela molto utile e produttiva ai fini della scrittura del programma. Questa gerarchia si può rappresentare esattamente come un albero genealogico. I benefici portati da questa pratica sono vari.

Immaginiamo un programma che faccia largo uso delle finestre, quindi creiamo una classe che descrive i vari comportamenti di una finestra: la riduzione a icona, lo spostamento, l'ingrandimento ecc. Noi sappiamo dall'uso del computer che non c'è solamente un tipo di finestra: ci sono le MessageBox, le finestre di dialogo, le finestre di editing, ecc. quindi dovremmo creare una classe per ogni tipo di finestra e ridefinire i vari comportamenti per ogni tipo di finestra; ma se invece creaiamo una classe per ogni finestra che eredita da una classe principale di una finestra generica, eviteremo di definire i comportamenti principali per ogni finestra.

Quindi un oggetto2 che eredita da oggetto1 potrà usufruire di tutti i suoi campi e metodi.

Da qui appare chiaro il perché usare l'ereditarietà. Ecco un esempio in codice:

```
TYPE TBox1 = OBJECT
                Altezza, Lunghezza, Larghezza: integer;
                PROCEDURE Disegna;
                PROCEDURE Imposta(alt, lung, larg: integer);
             END;
     TBox2 = OBJECT (TBox1)    {TBox2 eredita TBox1}
                PROCEDURE ApriScatola;
                PROCEDURE ChiudiScatola;
             END;
```

Il polimorfismo

Il polimorfismo è un concetto un po' più avanzato dei precedenti, ma comunque importantissimo. Polimorfismo vuol dire "molte forme", e diciamo subito che si riferisce a una procedura o una funzione. Facciamo subito un esempio: supponiamo di voler creare un programma per disegnare delle figure piane (triangoli, rettangoli, circonferenze ecc.). Per prima cosa creiamo un oggetto comune, per esempio `figurabidimensionale`, e supponiamo che contenga una procedura di nome `Disegna`. La dichiareremo `virtual`, ovvero non la definiamo nell'oggetto `figurabidimensionale`, ma la definiamo negli oggetti che ereditano questo. Comunque i metodi virtuali sono un concetto che esporremo più avanti.

Da questo oggetto erediatiamo altri oggetti di nome `triangolo`, `rettangolo`, `circonferenza`, e definiamo il metodo `Disegna` per ogni oggetto (che ovviamente avrà una diversa implementazione per ciascuno).

Adesso, supponiamo che una procedura più avanti si serva del metodo `draw`, ma non specifica quale oggetto stiamo utilizzando. Per esempio:

```
PROCEDURE UnaProcedura(figura:figurabidimensionale);
BEGIN
   ...
   ...
   ...
   figura.Draw;
   ...
   ...
   ...
END;
```

In questo esempio il metodo Draw non sarà quello di `figurabidimensionale`, ma quello della classe figlia che eredita, anche perché in `figurabidimensionale` il metodo non è definito.

Questa pratica è chiamata **polimorfismo**.

Come creare l'istanza di un oggetto

Quando definiamo un oggetto con la clausola `TYPE` in realtà l'oggetto non è stato creato, ma ne è stata definita la struttura, cioè la **classe**. Creare un'istanza di un oggetto vuol dire allocarlo nella memoria heap. La memoria heap è una memoria con una struttura ad alberi che, a differenza dello stack, vi si accede tramite i puntatori, ovvero variabili speciali che contengono un indirizzo di memoria. Quindi un oggetto creato, o meglio, allocato, ha sempre un puntatore per il riferimento.

Bene, nel Pascal, per motivi di compatibilità, è possibile riferirsi a un oggetto creando una variabile con la clausola `VAR`, ma comunque questo serve solo per lanciare l'oggetto principale del programma che una volta avviato, sarà meglio utilizzare i puntatori per riferirci a un oggetto. Quindi riprendendo l'esempio sopra, se voglio creare come primo passaggio del programma un oggetto TBox1, userò la seguente sintassi:

```
VAR Box1:TBox1;
BEGIN
   ...
   ...
   ...
END.
```

Per creare l'istanza di un oggetto utilizzando un puntatore, prima devo definire un puntatore all'oggetto-tipo:

```
PBox2: ^TBox2;
```

A questo punto possiamo utilizzare la nuova versione della procedura del Turbo Pascal `new`. La funzione primaria di questa procedura è quella di creare una variabile dinamica e consentire al puntatore P passato come parametro di puntare su di essa. Col Turbo Pascal 6.0 la procedura, sebbene ancora valida, è stata trasformata ìn funzione. Il suo compito è di rilasciare come valore un puntatore all'oggetto. La sua sintassi è questa:

```
P:= new(PBox2, Init);
```

dove P è un puntatore generico e `Init` è il costruttore della classe TBox2 (i costruttori saranno trattati più avanti).

I modificatori public, private e protected

Nella programmazione orientata agli oggetti i campi e i metodi possono essere dichiarati **public** e **private**. Esistono anche altri modificatori di campi e metodi, tipici di ogni linguaggio; il Turbo Pascal prevede anche il modificatore **protected**.

I modificatori public e private

In una classe possiamo decidere se i campi e i metodi sono accessibili dall'esterno. Per esterno si intende tutto il codice che non fa parte di procedure e funzioni di un determinato oggetto (o meglio, classe). Dichiarando `public` un campo (o un metodo) di un oggetto, anche altri oggetti potranno utilizzare il campo (o il metodo) dichiarato public. Al contrario, se dichiaro un campo o un metodo come `private`, possiamo utilizzare lo stesso solamente nell'oggetto in cui è stato dichiarato.

In Turbo Pascal 7.0 se viene omessa la dichiarazione `public` o `private`, verrà assegnato, di default, il modificatore `public`.

La scelta di dichiarare `public` o `private` un campo o un metodo sta tutto nella scelta del programmatore per motivi di convenienza. Solitamente, sono i campi di un oggetto ad essere dichiarati `private`, mentre i metodi, che agiscono il più delle volte sui campi, vengono dichiarati `private`.

Un esempio:

```
TYPE MyObject = OBJECT
                   PUBLIC:
                     {                             }
                     {campi e modificatori public }
                     {                             }
                   PRIVATE:
                     {                             }
                     {campi e modificatori private}
                     {                             }
                  END;
```

Il modificatore protected

Il modificatore `protected` è un modificatore speciale. Possiamo definirlo come una via di mezzo tra il modificatore `public` e il modificatore `private`, infatti dichiarando un campo o un metodo come `protected`, lo stesso sarà utilizzabile solamente dall'oggetto in cui è stato dichiarato **e dai suoi dicendenti**, ovvero da tutti gli altri oggetti che ereditano.

La sintassi è analoga a quanto abbiamo visto con `private` e `protected`.

Costruttori e distruttori

Quando un oggetto viene creato, vale a dire allocato nella memoria heap e avente un puntatore come riferimento, dovrà essere opportunamente inizializzato affinché il programma funzioni come voluto. Un oggetto, dopo essere stato creato, se non servirà più al programma potrà essere distrutto, liberando le risorse occupate dall'oggetto stesso e, se necessario, eseguire le dovute azioni prima della sua eliminazione. La programmazione orientata a gli oggetti fornisce due metodi ad hoc per la costruzione e la distruzione degli oggetti, ovvero i cosiddetti costruttori e distruttori.

I costruttori

Una macchina quando deve essere accesa e partire deve essere "manovrata" bene affinché tutto funzioni secondo la nostra intenzione. Se non è ingranata la prima marcia e se non viene premuta la frizione la macchina potrebbe non partire o spegnere il motore a causa del numero basso di giri non appena viene rilasciata la frizione.

Anche un oggetto, in un programma, se non inizializzato correttamente, potrebbe divenire inutile o malfunzionante ai fini del programma. Il Pascal per riferirsi a un metodo che funge da costruttore, utilizza la parola chiave `CONSTRUCTOR`. Supponiamo di creare un oggetto che stampa su un ambiente testuale la famosa frase "Hello, world!". Inizializziamo l'oggetto con la posizione dove dovrà essere stampata la frase, e poi utilizziamo il metodo `Stampa` per stampare la frase.

```
USES crt;
TYPE HelloWorld= Object
                   PRIVATE
                   riga,colonna:byte;
                   PUBLIC
                   Constructor Init(rig,col:byte);
                   Procedure Stampa;
                 End;

Constructor HelloWorld.Init;
BEGIN
   riga:=rig;
   colonna:=col;
END;
Procedure HelloWorld.Stampa;
BEGIN
   gotoxy(riga,colonna);
   write('Hello, world!');
END;
VAR CiaoMondo:HelloWorld;
BEGIN
   ClrScr;
   CiaoMondo.Init(30,5);
   CiaoMondo.Stampa;
   readln;
END.
```

Il costruttore `Init` inizializza i campi `riga` e `colonna` secondo i parametri passati, `Stampa` sposta il cursore nel punto indicato dai campi `riga` e `colonna`, e stampa finalmente la frase.

I costruttori in un oggetto non sono ovviamente obbligatori, ma se ne raccomanda l'uso.

I distruttori

Analogamente ai construttori, i distruttori fanno quel che necessita non appena l'oggetto viene distrutto, anche se, la funzione principale di un distruttore è quella di liberare la memoria e renderla disponibile ad altre parti del programma. La parola chiave del Pascal che denota un distruttore è `DESTRUCTOR`

Come esempio possiamo ancora considerare il precedente, definendo un distruttore che oltre a liberare le risorse e a cancellare l'istanza dell'oggetto, cancella la frase che era stata scritta nello schermo.

```
USES crt;
TYPE HelloWorld= Object
                   PRIVATE
                   riga,colonna:byte;
                   PUBLIC
                   Constructor Init(rig,col:byte);
                   Procedure Stampa;
                   Destructor Done;
                 End;

Constructor HelloWorld.Init;
BEGIN
   riga:=rig;
   colonna:=col;
END;
Procedure HelloWorld.Stampa;
BEGIN
   gotoxy(riga,colonna);
   write('Hello, world!');
END;
Destructor HelloWorld.Done;
BEGIN
   gotoxy(riga,colonna);
   write('             ');
END;
VAR CiaoMondo:HelloWorld;
BEGIN
   ClrScr;
   CiaoMondo.Init(30,5);
   CiaoMondo.Stampa;
   ReadLn;              {Attende che premiamo INVIO}
   CiaoMondo.Done;      {Distruzione, lo schermo è tutto nero}
   ReadLn;
END.
```

Override, metodi virtuali e classi astratte

Verranno ora introdotti alcuni concetti avanzati del paradigma OOP.

Override

Nella programmazione orientata agli oggetti è possibile ridefinire un metodo che è stato ereditato dall'oggetto padre. Il metodo ovviamente dovrà avere lo stesso nome e lo stesso numero e tipo di parametri.

Facciamo subito un esempio:

```
TYPE TA = Object
            Procedure StampaMsg;
         End;
     TB = Object (A)
         End;
Procedure A.StampaMsg;
BEGIN
   writeln("Messaggio del metodo originale.");
END;
Procedure B.StampaMsg;                {Override}
BEGIN
   writeln("Questo invece è il messaggio del metodo ridefinito.");
END;
VAR A:TA;
    B:TB;
BEGIN
   A.StampaMsg;   {metodo originale}
   B.StampaMsg;   {metodo ridefinito}
   readln;
END.
```

Molte volte però si ha l'esigenza di creare un'estensione del metodo originale, in modo che possa fare altro e di più. Il Turbo Pascal ammette la parola riservata `inherited` per includere nel metodo la routine compresa nel metodo originale.

Riferendoci all'esempio di cui sopra, possiamo modificarlo per mostrare un possibile utilizzo della parola chiave **inherited**.

```
TYPE TA = Object
            Procedure StampaMsg;
         End;
     TB = Object (A)
         End;
Procedure A.StampaMsg;
BEGIN
   writeln("Messaggio del metodo originale.");
END;
Procedure B.StampaMsg;                {Override}
BEGIN
```

```
   inherited StampaMsg;         {qui il compilatore include la routine
del metodo dell'oggetto padre}
   writeln("Questo invece è il messaggio del metodo ridefinito.");
END;
VAR A:TA;
    B:TB;
BEGIN
   B.StampaMsg;   {metodo ridefinito}
   readln;
END.
```

L'effetto del programma sopra sarà di stampare entrambi messaggi, ovvero quello della funzione originale e della funzione ridefinita, grazie all'istruzione

```
inherited StampaMsg
```

.

I metodi virtuali e le classi astratte

Un metodo è chiamato virtuale quando non è definito nell'oggetto dove è stato dichiarato. Quindi la sua presenza sarà appunto virtuale.

Una classe che contiene metodi virtuali è chiamata classe astratta poiché, non avendo i metodi virtuali definiti, la stessa non può essere istanziata, cioè non possiamo creare un oggetto che contiene metodi astratti. Una classe astratta servirà come modello per altre classi che erediteranno da lei.

Il Turbo Vision

Il Turbo Vision è un framework per ambiente DOS testuale sviluppato dalla Borland per il Turbo Pascal e il Turbo C. Un framework è innanzitutto una libreria, ma anche uno strumento che ci aiuta nella stesura del programma e ci consente di gestire meglio le caratteristiche di un programma.

Con Turbo Vision è possibile creare applicazioni gradevoli dal punto di vista sia grafico che funzionale e si presenta più intuitivo all'utente; ci consente infatti di gestire i contenuti tramite le finestre, di creare finestre di dialogo con dei controlli utente, creare barre di stato e menù e tanto altro ancora.

Il Turbo Vision è interamente orientato agli oggetti ed è presente in una collezione di file TPU (Turbo Pascal Unit) realizzata dalla Borland. Come primo esempio dimostrativo possiamo compilare il seguente codice:

```
uses App;

type
   TMyApp = object(TApplication)
            end;

var MyApp : TMyApp;

begin
  MyApp.Init;
  MyApp.Run;
```

```
  MyApp.Done;
end.
```

Il semplicissimo programma non fa che ereditare dalla classe TApplication, richiamando il costruttore Init, il metodo Run e il distruttore Done.

Graficamente disegna uno sfondo con caratteri speciali tipico del Turbo Vision e una barra di stato con la scritta "Alt+X Exit". Per uscire sarà sufficiente premere Alt+X oppure cliccare nella scritta in basso a sinistra.

Non approfondiremo più di tanto il Turbo Vision, anche perché esula dagli scopi di questo libro. Per maggiori informazioni su TV siete rimandati all'apposito libro.

Strumenti

Compilatori

I compilatori eseguono la traduzione del linguaggio Pascal in linguaggio macchina, creando cosi un file eseguibile. Affianco a quelli a pagamento offerti dalla Borland sono sempre più diffusi compilatori gratuiti e open-source, sviluppati inizialmente per ambienti Linux.

- A pagamento
  - Borland Pascal
  - Borland Delphi
- Open Source
  - Free Pascal
  - GNU Pascal
  - DEV Pascal

Problemi di compatibilità

L'uso di differenti compilatori può far sorgere alcuni problemi di incompatibilità tra versioni, non tanto comunque per quanto riguarda il linguaggio vero e proprio.

Ad esempio, il compilatore di Delphi (che serve per creare applicazione normalmente basate su un'interfaccia grafica) necessita della direttiva di compilazione

```
{$APPTYPE CONSOLE}
```

da porre subito dopo l'intestazione del programma, la quale specifica che il programma è da eseguire in un'interfaccia solo testo della console.

Inoltre, sempre per quanto riguarda l'uso di Borland Delphi, non è necessario eseguire alcune operazioni come la cancellazione dello schermo, in quanto ciò viene fatto in automatico all'avvio del programma da esso compilato.

A seconda della versione più o meno recente del compilatore, il tipo di dato *integer* supporta una gamma di valori più vasta di -32.768...32.767

IDE

IDE è l'acronimo per *Integrated Development Environment*, che significa ambiente di sviluppo integrato.
L'IDE è un particolare software che facilita il programmatore nella stesura, nella compilazione e nella distribuzione del programma. Un tipico esempio di aiuto offerto da un IDE è l'evidenziazione della sintassi e l'individuazione di semplici errori di sintassi prima della compilazione.

- A pagamento
  - Borland Delphi
  - Borland Turbo Pascal
- Open Source
  - Lazarus [1]
  - Dev-Pas [2]
  - FreePascal [3]

Da notare che Borland Delphi e Lazarus sono IDE di Delphi, ovvero un'evoluzione del Pascal orientata ad oggetti; riescono tuttavia a compilare senza problemi il Pascal normale.

Compilare i propri sorgenti

Abbiamo visto in precedenza il concetto di compilatore: la sua funzione è quella di tradurre il linguaggio Pascal in un linguaggio comprensibile al computer (linguaggio macchina).
Il processo della compilazione può avere esito positivo come può fallire a causa di **errori di compilazione**, cioè errori di sintassi o di grammatica nel linguaggio Pascal. Questi errori sono generalmente segnalati dai compilatori con messaggi del tipo:

```
Error on line 17: ';' expected but variable found
```

Questo è un errore che vi potrà capitare molto spesso, e segnala la mancanza di un punto e virgola alla fine di un'istruzione.
Il compilatore non conclude la compilazione fino a che il listato non presenterà errori.

Su sistemi operativi Linux una buona soluzione consiste nell'uso di Free Pascal. Per compilare un sorgente sono sufficienti pochi comandi di console; innanzitutto è necessario spostarsi nella directory in cui è presente il sorgente, successivamente basta eseguire le istruzioni:

```
> fpc nome_sorgente.pas
... il programma fornisce indicazioni sul compilatore, sul file
    e sugli eventuali errori di compilazione riscontrati ...
```

Potremo eseguire il programma digitando:

```
> ./nome_sorgente
```

Su Windows l'uso di Free Pascal da riga di comando risulta identico dal prompt dei comandi tranne per l'esecuzione del file, che viene fatta nella forma

```
> nome_sorgente.exe
```

L'uso degli IDE semplifica molto questi processi. Generalmente è possibile individuare nella barra dei menu di questi programmi voci come Compile -> Compile oppure Run -> Run (esegue il programma) oppure Compile -> Run.

Alcuni esempi commentati

Questo è un programma che, dato un numero, ne calcola la radice quadrata (senza l'uso della funzione `sqrt`, ovviamente) usando una regola spiegata durante l'esecuzione del programma.
Il cuore dell'algoritmo è un ciclo ripeti... sino a quando che, oltre a calcolare le approssimanzioni, stampa anche una tabella passo per passo dei calcoli effettuati

```
program radq;
uses crt;
 {$APPTYPE CONSOLE} {in alcuni compilatori è superfluo}
 const eps=1E-10;
 var  num, app: real;
begin
 clrscr;
 write('Questo programma si basa sulla regola di Newton');
 writeln(' riguardante le radici quadrate, secondo la quale ');
 writeln;
 writeln('     se app e'' un'' approssimazione della radice quadrata di
 num');
 writeln('     allora (num/app + app)/2 e'' un''approsimazione
migliore.');
 writeln;
 write('Il programmera continuera'' a calcolare un''approssimazione
migliore');
 writeln(' sino a quando non varra''' );
 writeln;
 writeln('     |num/app^2 - 1| < 10^-10');
 writeln;
 writeln('Introdurre il valore del numero num di cui si vuole calcolare
 la radice quadrata e della prima approssimazione app');
 writeln;
 write('     '); readln(num);
 write('     '); readln(app);
 writeln;
 if num < 0 then
  writeln(num:10:2, ' non ha radice quadrata reale')
 else
 begin
  {stampa una piccola tabella}
  writeln('         app                          app^2');
  writeln;
  repeat
   writeln(app:20:10, app*app:30:10); {stampa la riga}
   app:=(num/app +app)/2; {calcola la successiva approssiamazione}
  until abs(num/ sqr(app) -1) < eps; {se il valore non differisce troppo da quello reale}
  writeln;
  writeln(app:20:10, app*app:30:10); {stampa l'ultima approssimazione
contente il valore definitivo della radice quadrata}
```

```
 end;
 readln; {in alcuni compilatori è superfluo}
end.
```

Il seguente esempio è un programma che permette invece di simulare il lancio di un oggetto con velocità e posizione iniziale scelte dall'utente; per fare ciò si ricorre all'uso della libreria graph, a cui è stata dedicata una sezione nel corso del libro.
Come per il programma precedente, bisognerà adattare alcune opzioni in base al compilatore in uso.
Il cuore del programma è qui la procedura lancio, che calcola man mano le posizioni dell'oggetto lanciato e le disegna sullo schermo grafico.

```
program lanci;
uses crt, graph;
{i tipi word e double sono estensioni dei tipi di dati rispettivamente
integer e real}
{funzionano in modo analogo ma hanno un range di dati diverso}
var t,v0,x0, y0:double; a:real;
    s,m:smallint;
    c:word;
    r:char;
const g = 9.8;

{questa procedura serve a cambiare sempre i colori}
procedure move_color (pos:smallint);
begin
    c:=c+pos;
    setcolor(c mod getmaxcolor);
end;


procedure lancio (v0, x0, y0 :double; a:real; delay_time:integer);
{longint è un'estensione degli integer ma con più range}
var y,x, vx, vy, t:double;
    xi, yi:smallint;
begin
    a:=pi() * a / 180;
    vx := v0 * cos(a);
    vy := v0 * sin(a);
    {è necessario troncare i valori perché la grafica funziona solo con
 valori interi}
    moveto(trunc(x0), getmaxy-1-trunc(y0));

    move_color(1);

    repeat
    begin
     y:=vy * t-( g/2 * t* t)+y0;
     x:=vx * t+x0;
```

```
     {0,0 corrisponde all'angolo in alto a sinistra, noi vogliamo
quello in basso a sinistra}
     {per questo invertiamo il valore della y secondo getmaxy}
     yi:=trunc(getmaxy-y);
     xi:=trunc(x);
     move_color(1);
     lineto(xi,yi);
     move_color(-1);
     circle(xi,yi,20);
     t:=t+0.25;
     delay(delay_time);
    end;
    until (yi > getmaxy); {fino a quando la "pallina" non esce dallo
schermo}

    writeln('Lancio terminato. Premere un tasto per continuare');
    readkey;
end; {lancio}

begin
    s:=detect;
    initgraph(s,m,''); {ovviamente bisogna cambiare la directory}
    directvideo:= true;

    repeat
    clrscr;
    begin
     writeln('       SIMULAZIONE DI LANCI INCLINATI');
     {stampa un piccolo menu}
     writeln; writeln;
     writeln('   L. Nuovo lancio');
     writeln('   C. Cancella schermo grafico');
     writeln('   E. Esci dal programma');
     writeln;writeln;
     write('               Selezionare un''opzione -> '); readln(r);
     {le diverse opzioni...}
     case r of
      'L', 'l':
      begin
         clrscr;
         writeln('Introdurre i valori della velocita'' iniziale v0
(espressa in m/s)');
         writeln('della misura in gradi dell''angolo di inclinazione
del lancio dal suolo');
         writeln('e della posizione (x, y) iniziale del lancio');
         write('  v0      = '); readln(v0);
         write('  ang (ø) = '); readln(a);
```

```
         write('  x0       = '); readln(x0);
         write('  y0       = '); readln(y0);
         {legge le opzioni e chiama la procedura lancio}
         lancio(v0, x0,y0, a, 30);
      end;
      'c', 'C':
      begin
         {per cancellare lo schermo chiudiamo e riapriamo la sessione
grafica}
         closegraph;
         s:=detect;
         initgraph(s,m,'');
         directvideo:= true;
      end;
     end; {case of}
    end; {until}
    until (r = 'e') or (r  = 'E');
    closegraph;
end.
```

Note

[1] http://lazarus.freepascal.org
[2] http://www.bloodshed.net/dev/pascal.html
[3] http://www.freepascal.org

Esercizi

Parte 1

In questa prima parte si svolgeranno esercizi molto semplici per avvicinarsi al linguaggio Pascal

Esercizio 1

Hello World

Soluzione

```
program hello;
uses crt;
begin
 writeln("Hello World");
end.
```

Esercizio 2

Dati due numeri (inseriti dall'utente) se ne calcoli la somma

Soluzione

```
program somma;
uses crt;
var a, b:Integer;
begin
 writeln("a + b = c");
 write("a: ");
 readln(a);
 write("b: ");
 readln(b);
 write("c = ");
 writeln(a+b);
 readln;
end.
```

Esercizio 3

Scrivere un programma che consenta ad un utente di calcolare la somma dei primi n numeri naturali (con n inserito da tastiera).

Soluzione

```
program somma;
uses crt;
var n, temp, i:Integer;
begin
 writeln("Inserire il numero n");
```

```
 readln(n);
 temp :=0;
 for i:=1 to n do
  temp:=temp + i;
 writeln("Totale: " + temp);
end.
```

Parte 2

In questa parte si svolgeranno esercizi di media difficoltà per comprendere le strutture dati e i sistemi di sviluppo tipici del mondo imperativo

Parte 3

In questa parte si svolgeranno esercizi tipici di gestione dell'I/O.

Esercizio 1

Copia file: il programma apre un file in lettura, legge riga per riga e ne riscrive il contenuto in un altro file.

Soluzione

```
Program copia;

var fin, fout: TEXT;
   N, i: integer;

begin
   assign(fin, 'input.txt');
   reset(fin);
   assign(fout, 'output.txt');
   rewrite(fout);

   readln(fin, N);
   for i := 1 to N do
       writeln(fout, i);

   close(fin);
   close(fout);
end.
```

Parte 4

In questa parte si svolgeranno esercizi complessi che mostrino cosa si possa fare con Pascal base per ottenere risultati anche molto complessi

Parte 5

In questa parte si svolgeranno esercizi di Pascal ad Oggetti

Questo modulo è solo un abbozzo. a migliorarlo secondo le convenzioni di Wikibooks

Fonti e autori delle voci

Fonte:: http://it.wikibooks.org/w/index.php?oldid=108681 *Autori:*: Diablo, G4, Otrebla86, Pietrodn, Ramac

Pascal *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=222330 *Autori:*: Daniele, Diablo, Flash, Gianfranco, LoStrangolatore, Marcozampini, Massimiliano Lincetto, Mmo, Otrebla86, Peppe, Pietrodn, Pil56, Ramac, Scmuzhla, Wim b, Xno, 43 Modifiche anonime

Introduzione *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=220252 *Autori:*: Diablo, G4, KaeZar, Ramac, The Doc, 2 Modifiche anonime

Algoritmi *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=218308 *Autori:*: Frigotoni, Ramac, 4 Modifiche anonime

Variabili *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=167584 *Autori:*: Diablo, G4, Otrebla86, Ramac, WK, 4 Modifiche anonime

Input e output *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=110524 *Autori:*: Ciampix, Diablo, G4, Ramac, 3 Modifiche anonime

Tipi di dati e operatori *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=220090 *Autori:*: Diablo, Flash, G4, Lemming, Pietrodn, Ramac, The Doc, 11 Modifiche anonime

Gestione avanzata dei dati *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=219444 *Autori:*: Ciampix, Crimer, Dedot, Ramac, 2 Modifiche anonime

Commenti *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=95077 *Autori:*: Diablo, G4, Ramac, The Doc, Wim b

Librerie e funzioni predefinite *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=188310 *Autori:*: *kinny*, Diablo, Flash, G4, Otrebla86, Pietrodn, Ramac, The Doc, 6 Modifiche anonime

Istruzioni di controllo *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=216301 *Autori:*: Dedot, Diablo, G4, LoStrangolatore, Max91, Pietrodn, Ramac, The Doc, Wim b, 9 Modifiche anonime

Array *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=192414 *Autori:*: Dedot, Diablo, Fale, G4, Ramac, 6 Modifiche anonime

Metodo top-down, procedure e funzioni *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=198578 *Autori:*: Dedot, G4, Link, LucAndrea, Ramac, 9 Modifiche anonime

Programmazione ad oggetti *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=166391 *Autori:*: Ciampix, Otrebla86, Ramac, 1 Modifiche anonime

Concetti fondamentali *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=207463 *Autori:*: Cekkins, Otrebla86, Pietrodn, Ramac, 4 Modifiche anonime

I modificatori public, private e protected *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=165782 *Autori:*: Otrebla86

Costruttori e distruttori *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=220887 *Autori:*: Otrebla86, 2 Modifiche anonime

Override, metodi virtuali e classi astratte *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=176509 *Autori:*: Otrebla86

Il Turbo Vision *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=176507 *Autori:*: Otrebla86

Strumenti *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=187709 *Autori:*: Diablo, G4, Pietrodn, Ramac, 6 Modifiche anonime

Esercizi *Fonte:*: http://it.wikibooks.org/w/index.php?oldid=215323 *Autori:*: Ciampix, LoStrangolatore, Mmo, Wikitanvir, 3 Modifiche anonime

Fonti, licenze e autori delle immagini

File:Blaise pascal.jpg *Fonte:*: http://it.wikibooks.org/w/index.php?title=File:Blaise_pascal.jpg *Licenza*: Public Domain *Autori:*: Anarkman, Crux, Deadstar, Killiondude

Immagine:Wikipedia-logo.png *Fonte:*: http://it.wikibooks.org/w/index.php?title=File:Wikipedia-logo.png *Licenza*: logo *Autori:*: version 1 by Nohat (concept by Paullusmagnus);

Image:Monte-Carlo method pi.svg *Fonte:*: http://it.wikibooks.org/w/index.php?title=File:Monte-Carlo_method_pi.svg *Licenza*: Public Domain *Autori:*: Ramac

Image:Wikipedia-logo.svg *Fonte:*: http://it.wikibooks.org/w/index.php?title=File:Wikipedia-logo.svg *Licenza*: logo *Autori:*: Tcrow777.

Immagine:Wiki letter w.svg *Fonte:*: http://it.wikibooks.org/w/index.php?title=File:Wiki_letter_w.svg *Licenza*: GNU Free Documentation License *Autori:*: Jarkko Piiroinen